DEC X.500 Directory Service

OSI-Abstract-Data Manipulation

Order Number: AA-PXVFA-TE

Revision/Update Information: 1.0

Digital Equipment Corporation
Maynard, Massachusetts

First Printing, March 1993

Please complete the Reader's Comments page at the end of the book. It will help us to keep improving our documentation.

This document was prepared using VAX DOCUMENT, Version 2.1.

Contents

3 Using the OSI-Abstract-Data Manipulation API

Part II Reference

4 Object Management Package

5 OSI-Abstract-Data Manipulation Functions

6 OSI-Abstract-Data Manipulation Syntaxes

7 Object Management Data Types

Tables

Preface

Purpose of this Guide

This guide describes Digital's OSI-Abstract-Data Manipulation Application Program Interface, and provides reference information for developers of applications that use it.

Throughout this guide, OSI-Abstract-Data Manipulation is abbreviated to OM, and Application Program Interface to API. Unless otherwise stated, OM API refers to Digital's implementation of the OM API specified by the X/Open Company Limited in conjunction with the X.400 API Association.

Structure of this Guide

This guide is divided into two parts. Part I introduces the terms and concepts used in OSI-Abstract-Data Manipulation, and with the aid of examples, explains how to use the OM API. You should read it in its entirety before you read Part II and any of the documents mentioned in the Related Documents section.

Part II gives reference information for the OM API, for example class definitions, routine descriptions, and data types.

Intended Audience

This guide is intended for programmers using Digital's OM API in conjunction with either the MAILbus 400 API or the Digital X.500 API (or both) to develop an application.

Prerequisites

This guide assumes that you have read the introductory chapters in the documentation for either the MAILbus 400 API or the X.500 API, and that you have a basic understanding of purposes for which your application will be using the OM API.

You must be familiar with and understand Open Systems Interconnection (OSI) terms and concepts.

You must also be familiar with the C programming language, which is the programming language supported by the OM API.

Related Documents

Before you start writing your application, make sure that you have the documentation for the other API (or APIs) that you are going to use:

- MAILbus 400 API documentation
- Digital X.500 API documentation

For background information on the MAILbus 400 Message Transfer Agent and an explanation of the concepts associated with a Message Handling System, refer to *MAILbus 400 MTA Introduction*.

For background information on Digital's X.500 Directory Service and an explanation of the concepts associated with a Directory Service, refer to the Digital X.500 documentation set.

The name of the X/Open specification to which the OM API conforms is *X/Open CAE Specification, OSI-Abstract-Data Manipulation API (XOM)*. It is available from:

X/Open Company Limited,
Apex Plaza,
Forbury Road,
Reading,
Berks, RG1 1AX,
United Kingdom.

Conventions

The following conventions are used in this book:

`this typeface`	Indicates an example of code
newterm	Indicates the introduction of a new term.
variable	Represents variables to type in commands or responses.
[0, 2^{32})	Means the range 0 to 2^{32} including the 0 but not the 2^{32}

Abbreviations

The following abbreviations are used in this guide:

ASN.1	Abstract Syntax Notation One
API	Application Program Interface
BER	Basic Encoding Rules
IM	Interpersonal Messaging
MH	Message Handling
OM	OSI-Abstract-Data Manipulation
OR	Originator/Recipient
OSI	Open Systems Interconnection
RD	Recipient Descriptor

Part I

Programming

This part introduces Digital's OSI-Abstract-Data Manipulation Application Program Interface (OM API) and explains how to use it. It contains three chapters:

- Chapter 1: this gives an introduction to the OM API.
- Chapter 2: this describes the key concepts in OSI-Abstract-Data Manipulation.
- Chapter 3: this describes how you can use the OM API to create and manipulate objects when building an application using the X.400 or the X.500 API, or both.

1
Introduction

This chapter gives an introduction to Digital's OSI-Abstract-Data Manipulation Application Program Interface (OM API).

The Digital OM API is an implementation of the OM API specified by the X.400 API Association and the X/Open Company Limited. It provides a standard interface for creating and manipulating abstract data items. In open systems, the definitive descriptions of abstract data items are given in Abstract Syntax Notation One (ASN.1), and the data items can therefore be large and complex. The OM API provides a model for manipulating data items regardless of their size and complexity, and in so doing, simplifies the task of programming.

The data items that the OM API allows you to manipulate are referred to throughout this manual as **objects**.

The Digital OM API is intended for use in writing applications with other Digital interfaces, for example, the MAILbus 400 API and the Digital X.500 API. When you are using the MAILbus 400 API, the Digital OM API enables you to create and manipulate objects required for a Message Handling application, for example, Messages and Non-delivery Reports. When you are using the Digital X.500 API, the Digital OM API enables you to create and manipulate objects required for use in an X.500 Directory Service, for example, Directory Entries and their attributes.

Because the Digital APIs are based on standard interfaces, the applications that you write using them can easily be adapted for use with other implementations of the standard interfaces.

1.1 Conformance

The Digital OM API conforms to the X/Open CAE Specification, OSI-Abstract-Data Manipulation API (XOM), produced in conjunction with the X.400 API Association, and published by the X/Open Company Limited (November 1991).

With respect to the requirements listed in the X/Open CAE Specification, the Digital OM API conforms as follows:

- Workspaces

  The Digital OM API associates the OM Package with each workspace it opens. It opens a workspace for each invocation of the Open function (MAILbus 400 API), and of the Initialize function (Digital X.500 API). The OM Package is described in Chapter 4.

- Aspects

  The Digital OM API implements all defined aspects of the standard OM API, with the options described in Section 1.2.

- Encoding Rules

  In its implementation of the Encode and Decode functions (see Section 3.8), the Digital OM API supports the ASN.1 Basic Encoding Rules (BER).

1.2 Digital's Implementation of the OM API

The X/Open CAE Specification leaves certain aspects of the interface to be decided by the implementor. These are as follows:

- The local character set representation and the precise mappings between the local character set and the various string syntaxes.

  The Digital OM API does not support local character sets when used with the MAILbus 400 API. When used with the Digital X.500 API, the Digital OM API supports T.61 as the non-local string syntax and ISO Latin 1 as the local string syntax. For more details refer to the X.500 API documentation.

- The precise definitions in C of the intermediate data types.

  In the Digital OM API, these are as follows:

```
typedef int          OM_sint;
typedef short        OM_sint16;
typedef long int     OM_sint32;
```

typedef unsigned	OM_uint;
typedef unsigned short	OM_uint16;
typedef long unsigned	OM_uint32;

- The length of the longest string that the Get function returns.

  When used with the MAILbus 400 API, the length of the longest string that the Get function returns is 1024 bytes. Any strings that are longer than this must be dealt with using the Read and Write functions.

  When used with the Digital X.500 API, there is no limit to the length of the string returned by the Get function.

- Whether the service reports an exception if an object supplied to it as an argument is not minimally consistent. An object is minimally consistent if:

  - The type of each of its attributes is specific to the object's class or one of its superclasses.
  - The number of values of each attribute is no greater than the class permits.
  - The syntax of each value is among those the class permits.
  - The number of bits, octets or characters in each string value is among those the class permits.

  Because the MAILbus 400 API and Digital X.500 API both check for the consistency of objects and report exceptions if necessary, the Digital OM API does not report an exception if an object supplied to it as an argument is not minimally consistent.

1.3 Digital's Extensions to the OM API

The Digital OM API provides macros for creating public objects dynamically. These macros are in the header file `xom.h` and allow you to set an empty descriptor to your required value, once you have declared a descriptor or descriptor list. The macros are as follows. Note that because they are proprietary extensions to the standard OM API, they have the prefix OMX.

- OMX_CLASS_DESC

  Sets an object's class descriptor to the specified class.

- OMX_BOOLEAN_DESC

  Sets a Boolean value descriptor for the specified type.

- OMX_ENUM_DESC

  Sets an enumerated value descriptor for the specified type.

- OMX_INTEGER_DESC

  Sets an integer value descriptor for the specified type.

- OMX_OBJECT_DESC

  Sets an object descriptor for the specified type.

- OMX_OM_NULL_DESC

  Sets the null descriptor that terminates the descriptor list.

- OMX_ATTR_TYPE_DESC

  Sets an attribute type descriptor using the specified object identifier string.

- OMX_ZSTRING_DESC

  Sets a string descriptor given a null (zero) terminated string using the specified type name.

- OMX_STRING_DESC

  Sets a string descriptor given the length and elements pointer and using the specified type name.

For examples of how these macros can be used, refer to Chapter 7.

1.4 Variable Names

When defining variables, avoid using names that are used in the Digital OM API. These begin with the prefixes OM_, OM_C, OM_S, om_, OMP, omp, OMX, and omx. Also avoid using names that are used in the MAILbus 400 API or the Digital X.500 API (refer to the relevant manual for details). If you define a variable that is already in use, you will see an error message when you attempt to link the application.

1.5 Reference Pages

If you are building your application on an ULTRIX system, you can get help on a particular function by using the `man` command to display the reference page for that function, for example:

```
% man om_get
```

If you forget the name of a function, or if you forget which functions are available, use the `apropos` command. The following example shows how to display a list of commands that include the word get; the `om_get` function will be included in the list.

```
% apropos get
```

The following example shows how to display a list of all the OM functions:

```
% apropos om_
```

The `man -k` command has the same effect as the `apropos` command.

1.6 Terminology

The OSI-Abstract-Data Manipulation terms and expressions that you need to be familiar with are described in Chapter 2.

The term **Service** refers to the Digital implementation of the OM API together with either the MAILbus 400 API or the X.500 API. The term **Client** refers to the application that uses the Service.

Throughout the rest of the manual, the term OM API refers to the Digital OM API, the term X.400 API to the MAILbus 400 API, and the term X.500 API to the Digital X.500 API.

2

OSI-Abstract-Data Manipulation Concepts

This chapter describes the key concepts in OSI-Abstract-Data Manipulation. You need an understanding of these concepts before using the OM API.

2.1 Objects

An **Object** is an item of OSI abstract data representing a real-world object or piece of information; for example, a person, an electronic mail message, a presentation address or a printer.

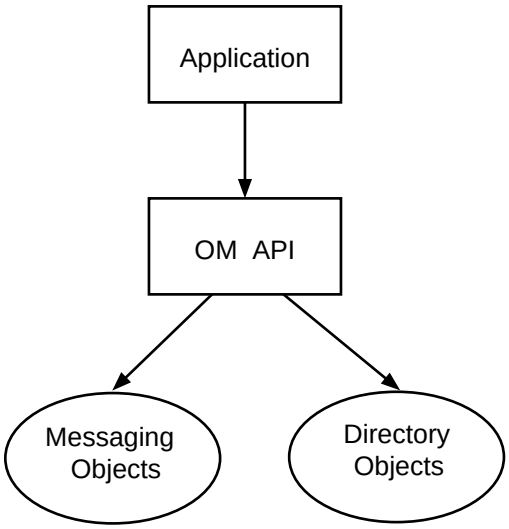
The X.400 API allows you to reference and maintain objects needed for message access, message transfer, and interpersonal messaging; that is, messaging objects. The X.500 API allows you to reference and maintain objects in an X.500 directory; that is, directory objects.

The following are examples of objects:

- Messaging objects:
  - Interpersonal Message
  - Delivered Report
- Directory objects:
  - Search-Result
  - Presentation-Address

Figure 2–1 shows the relationship between your application, the OM API, and messaging or directory objects.

Figure 2–1 Relationship between Application, OM and Objects



There are two types of object:

- Private objects
- Public objects

The OM API allows you to use both types in your application.

2.1.1 Private Objects

A **Private Object** is an object whose internal format is known only to the Service. A private object is identified by a pointer variable called a **Handle**. Your application can create and manipulate a private object by calling the OM API functions. You do not need to know anything about how the object is represented.

The facility to create private objects means that different Services can represent the same object in different ways. The X.400 API and the X.500 API, for example, each have their own internal representation for a Presentation Address.

A private object created in one workspace can be copied to a private object in another using the Copy function. The copy may have a different representation from the original. (For more information on workspaces, refer to Section 2.6.)

2.1.2 Public Objects

A **Public Object** consists of a data structure called a descriptor list (see Section 2.7). The descriptor list contains all the OM attribute values of the object.

There are two types of public object:

- Client-generated public objects

  These are public objects created by your application in storage allocated by the application.

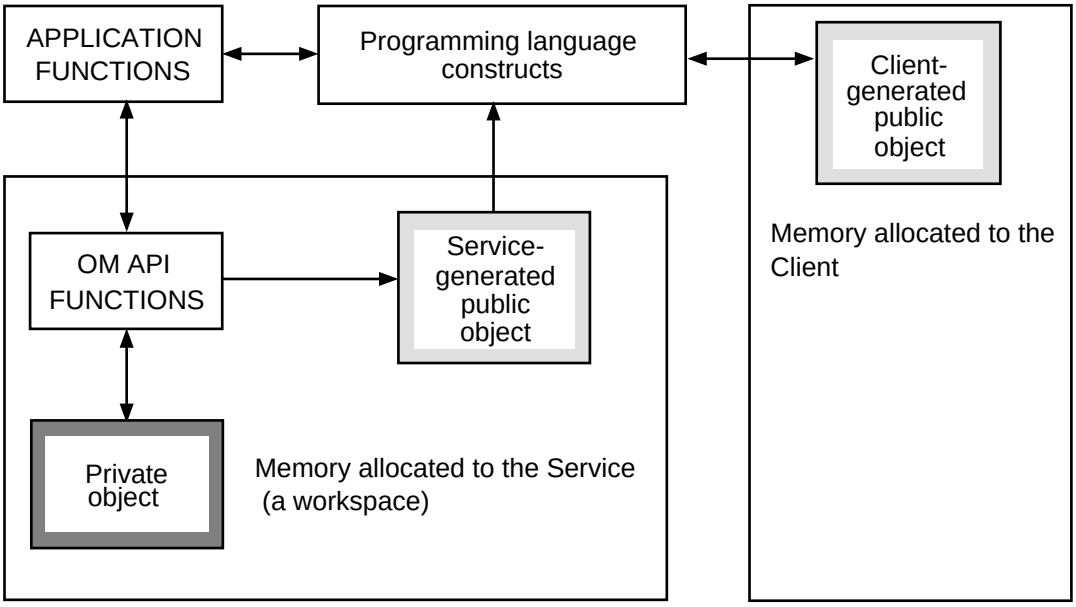
- Service-generated public objects

  These are public objects created by the Service in storage allocated by the Service. You can examine and delete service-generated public objects; you must not try to modify them.

Figure 2–2 shows private and public objects in relation to the OM API. The diagram shows that:

- The application can use the OM API functions to create and manipulate private objects and to generate service-generated public objects.
- The application can use programming language constructs to create and manipulate client-generated public objects and to read service-generated public objects.

Figure 2–2 Private and Public Objects



2.2 Attributes

An object consists of zero or more **attributes**. An attribute represents a specific item of information about the object. It consists of an attribute type, a syntax, and zero or more values.

The **attribute type** identifies the kind of information stored in the attribute. For example, the X.400 API attribute type Deferred Delivery Time is used to indicate that a message should not be delivered before a specified time. The attribute type can be thought of as the attribute name, and so the documentation might refer, for example, to the Deferred Delivery Time attribute.

The **attribute syntax** denotes the format of the information that can be stored in the attribute. For example, the Deferred Delivery Time attribute has the syntax String(UTC Time) (see Part II), and its values must therefore be in the format defined for representing times (UTC format).

The basic OM syntaxes are defined in Chapter 6. The documentation for the X.400 and X.500 APIs contain definitions of the syntaxes specific to those APIs.

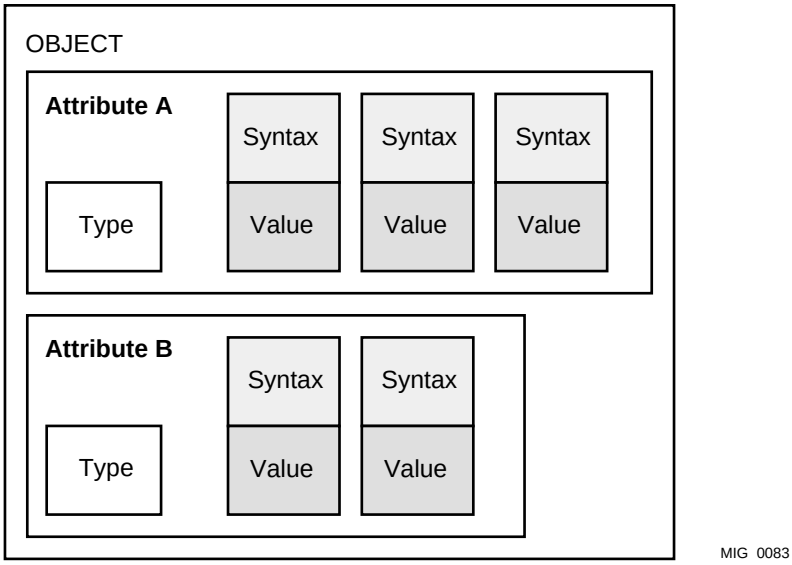
An **attribute value** is an item of information that you want to store in the attribute. For example, the Deferred Delivery Time attribute could contain the value 921211163120Z.

An attribute can be **single-valued** or **multi-valued**. A single-valued attribute has one value, whereas a multi-valued attibute can have more than one value.

The syntax of some attributes can be one of a specified set of syntaxes. Such attributes are called **set-valued** attributes. In the X.400 API, for example, the OR Address contains many attributes that can have values of either String(Printable) or String(Teletex) syntax, such as Postal Office Name. The values of a set-valued attribute do not all have to be of the same syntax. For example, Postal Office Name could have two values, one of syntax String(Printable), the other String(Teletex).

Figure 2–3 shows an object with two attributes, A and B. Attribute A has three values, and attribute B has two values.

Figure 2–3 An Object with Two Attributes



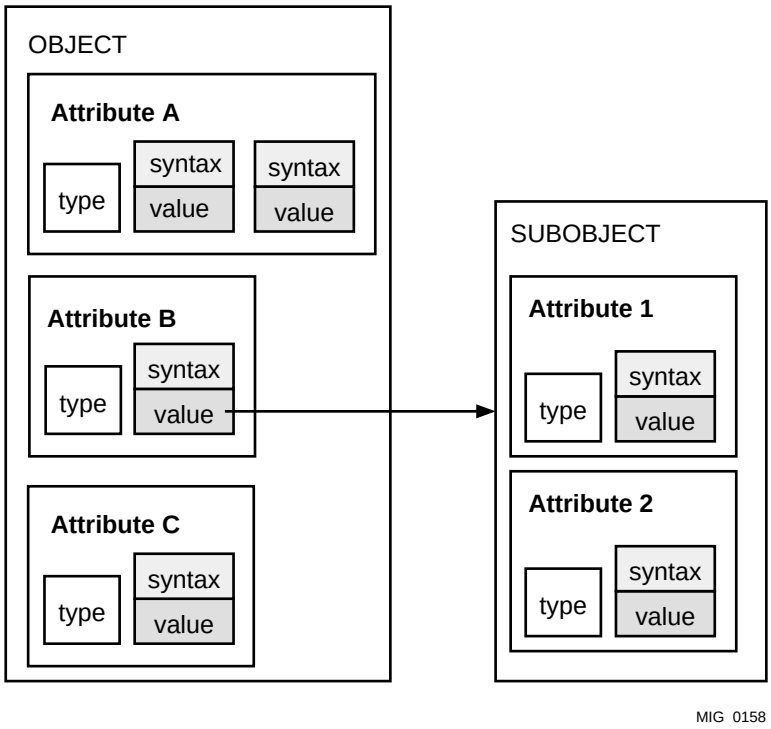
2.3 Subobjects

An object can have attribute values that are themselves objects. An object acting as an attribute value is called a **subobject**. Subobjects can be private or public objects, and can also have attributes that are objects. There is no limit to the number of levels of subobjects allowed within an object.

Figure 2–4 shows an object that has three attributes:

- Attribute A has two values.
- Attribute B has one value, which is a subobject with two attributes, 1 and 2, each of which has one value.
- Attribute C has one value.

Figure 2–4 A Subobject as an Attribute



Subobjects of a private object must also be private objects. You can use the Put function (see Table 3–1) to put a public object into an attribute of a private object. However, the public object is then copied and converted into a private object.

A client-generated public object can have subobjects that are either private or public (client-generated or service-generated). You add subobjects to client-generated public objects using programming language constructs (see Section 2.7).

A service-generated public object can have subobjects that are either service-generated public objects or private objects. This depends on the parameters you pass to the Get function (see Table 3–1) to produce the object.

2.4 Object Classes

Objects of similar structure or purpose are organized into groups called **classes**. Every object belongs to a class, and is known as an **instance** of that class. A class is characterized by a particular set of attributes.

The class of an object determines the attributes that may be present in the object, and may put constraints on those attributes. If, for example, class X has four attributes, 1, 2, 3 and 4, the class specification could impose the following constraints on those attributes:

- Attribute 1 is mandatory; it must have at least one value.
- Attribute 2 is optional; it can have zero or more values.
- Attribute 3 is set-valued; its value must be chosen from a set of possible values allowed by its syntax
- Attribute 4 is single-valued; it must have only one value. It has a string syntax associated with it, and the object class definition specifies the maximum length of the string value.

Chapter 4 of this manual, and the documentation for the X.400 and X.500 APIs, give details of the classes, attributes and attribute constraints defined for the APIs.

2.4.1 Class Hierarchy

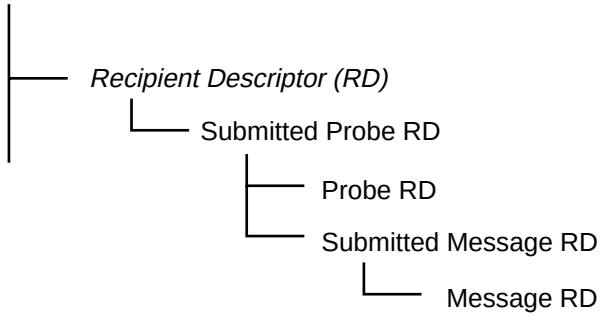
The OM API organizes classes into a hierarchy with a class called Object at its root. Every class except Object has a class immediately above it in the hierarchy, referred to as a **superclass**. It may also have a class (or classes) below it, referred to as a **subclass**.

Figure 2–5 shows a section of the X.400 API Message Handling class hierarchy. The diagram shows that the Recipient Descriptor (RD) class has one subclass, Submitted Probe RD, that the Submitted Probe RD class has two subclasses, Probe RD and Submitted Message RD, and that the Submitted Message RD class has one subclass, Message RD.

Important

An instance of a subclass has the attributes that characterize the class, but in addition inherits the attributes that characterize its superclasses.

Figure 2–5 Class Hierarchy



In the example in Figure 2–5, an instance of class Submitted Message RD (recipient descriptor) has the attributes of class Submitted Message RD. It also inherits the attributes of its superclass, Submitted Probe RD, including any attributes that an instance of class Submitted Probe RD inherits from its superclass, Recipient Descriptor. An instance of class Submitted Message RD therefore has the attributes of the Submitted Message RD, Submitted Probe RD and Recipient Descriptor classes.

Besides being an instance of its own class, an object is also an instance of all of its superclasses. If at any time in your application you need to use an object of a certain class, you can use an object of one of its subclasses instead. For example, in the hierarchy shown in Figure 2–5 you could use an object of class Message RD in place of an object of class Submitted Message RD. In the X.400 API, the MA Open function requires an object of class OR Address. However, you can call MA Open with an object of class OR Name because OR Name is a subclass of OR Address.

A class may be **concrete** or **abstract**. You can create an instance of a concrete class, but you cannot create an instance of an abstract class. Abstract classes exist as superclasses to enable attributes to be shared among several classes. In the example in Figure 2–5, Recipient Descriptor (RD) is an abstract class, as indicated by the italics.

The class hierarchy of the OM API is described in Chapter 4.

2.5 Packages

A **package** is a set of classes that are functionally related. The X.400 Message Handling (MH) package, for example, contains definitions for Messages, Reports, and Message Identifiers.

Each package is identified by an OSI Object Identifier. There are four X.400 packages and four X.500 packages available. Refer to the MAILbus 400 API documentation and the X.500 API documentation for details of the packages associated with each API. Refer to Chapter 4 of this manual for details of the OM package.

A **package closure** refers to the set of classes that need to be supported in order to be able to create all possible instances of all classes defined in a package. This is significant because a class can be defined to have an attribute whose value is an object of a class defined in another package. It then becomes necessary to support both packages in order to create an instance of such a class with that attribute in it. For example, the X.500 Basic Directory Contents package contains a class called Teletex-Term-Ident. This class has an attribute whose value is an object of a class called Teletex NBPs, which is defined in the X.400 Message Transfer package. Class Teletex NBPs is therefore said to be in the closure of the Basic Directory Contents package.

Package closures do not affect application developers using the X.400 or the X.500 API. Both these APIs are self-contained in that they include all the necessary class definitions.

2.6 Workspaces

The OM API maintains private objects and service-generated public objects in workspaces. A **workspace** is both an area of storage for objects in the closure of one or more packages that are associated with the workspace, and a definition of the OM functions that manipulate that data.

You can only create an instance of a class in a workspace if the package in which that class is defined is associated with the workspace. For example, to create an object of the X.500 API class DS-DN, the workspace must have the Directory Service package associated with it.

The OM package is automatically associated with every workspace that you open. The X.400 and X.500 APIs have their own mechanisms for associating with workspaces:

- Use the MA (Message Access) Open or the MT (Message Transfer) Open function to associate an X.400 package with a workspace. You may also associate Functional Units (small groups of related classes and functions)

with a workspace. Refer to the MAILbus 400 API documentation for details of Functional Units.

- Use the DS Initialize function to associate the X.500 Directory Service package with a workspace. Other packages are associated with a workspace using the DS Version function.

The implementation of the OM API functions may differ between workspaces. For example, the OM API implementation associated with an X.500 API workspace is different from that associated with an X.400 API workspace. However, this is transparent to the user of the OM API.

You cannot create a workspace which embraces packages from more than one of the APIs. You can, however, use an object from a workspace associated with one API as input to a function provided by another. If you use the X.500 API to build and access a directory service, for example, you could obtain an OR address from the directory and then add it to a message that you build using the X.400 API.

2.7 Descriptors and Descriptor Lists

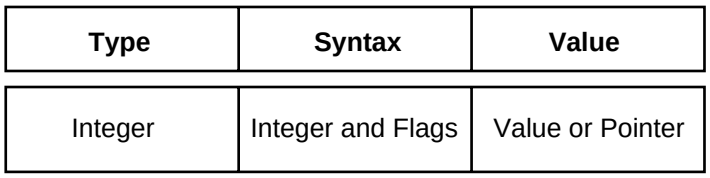
Descriptors and Descriptor Lists are data structures used to construct a public object. A private object has a single descriptor that identifies it (the Handle), but the internal construction is not apparent to the application. A public object is a list of descriptors referred to as a descriptor list.

A **descriptor** is a defined data structure that represents a single attribute. The structure has three components:

- A type, identifying the type of the attribute that the descriptor represents (the attribute name). This is an integer.
- A syntax, identifying the syntax associated with the attribute type that the descriptor represents. This is an integer and a set of flags.
- A value, identifying the value of the attribute that the descriptor represents. This can be one of the following:
  - The information stored in the attribute, if the type is a simple type; for example, Integer or Boolean.
  - A pointer to the information, if the type is an object or a string.

Figure 2–6 shows how a descriptor is constructed.

Figure 2–6 Components of a Descriptor



MIG0257

A **descriptor list** is an unordered set of descriptors that can represent several attribute types and values. It can be regarded as an array of the type OM_ descriptor (see Chapter 7). Descriptor lists can store single-valued and multi-valued OM attributes. Although attributes in a descriptor list are unordered, if an attribute is multi-valued, the order of the values must be preserved.

A single-valued attribute is represented by a single descriptor in a descriptor list. A multi-valued attribute is represented by several descriptors in a descriptor list, each with the same attribute type.

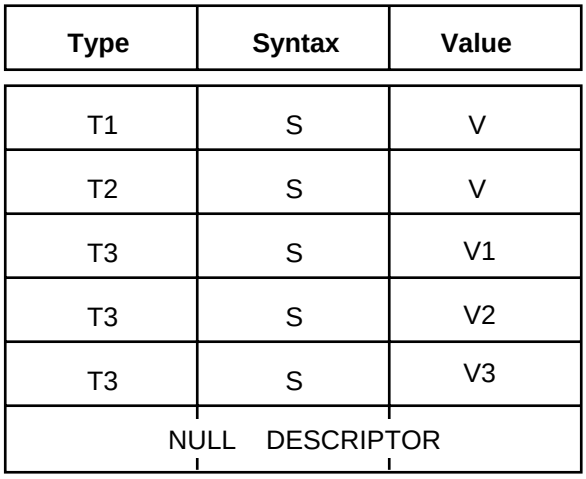
A single descriptor list can represent several single-valued and multi-valued attributes. In this case, descriptors of the same attribute type are always adjacent to each other in the descriptor list.

The descriptor describing the class of the object must, if present, be the first descriptor in the descriptor list. The constant OM_NULL_DESCRIPTOR denotes the end of the descriptor list.

Figure 2–7 shows how a descriptor list is constructed. The descriptor list contains the following descriptors:

- A class descriptor, T1, representing the class of the public object.
- A single descriptor, T2, representing a single-valued attribute in the public object.
- Three descriptors, T3, representing an attribute with three values in the public object.
- The null descriptor, signaling the end of the descriptor list.

Figure 2–7 Components of a Descriptor List



MIG0262

Part II of this guide includes descriptions of the Descriptor and Public Object data types.

3

Using the OSI-Abstract-Data Manipulation API

This chapter describes how you can use the OM API to create and manipulate objects that you need when building an application using the X.400 or the X.500 API, or both.

Table 3–1 shows you what the OM API allows you to do with objects, and gives the name of the function that you should use in each case.

Table 3–1 Object Management Functions

Task	Function Name	C Binding
Create a private object	Create	om_create( )
Assign values to attributes of a private object, or change existing attribute values of a private object	Put	om_put( )
Remove attribute values from a private object	Remove	om_remove( )
Create a public copy of the whole or part of a private object	Get	om_get( )
Make an exact but independent copy of a private object	Copy	om_copy( )
Determine whether an object is an instance of a named class	Instance	om_instance( )
Delete a private object or a service-generated public object	Delete	om_delete( )
Read a segment of a string from a private object	Read	om_read( )
Copy long string values from one private object to another	Copy Value	om_copy_value( )
Write a segment of a string to a private object	Write	om_write( )
Encode a private object	Encode	om_encode( )
Decode a private object	Decode	om_decode( )

Chapter 5 gives full information on how to call each of the OM functions.

Code Examples

The code examples in this chapter are fragments of C programs. They assume that the header file xom.h has been included and that a valid workspace has been set up by a call to the appropriate X.400 API or X.500 API function.

3.1 Creating an Object

Your application must be able to create objects to pass as arguments to the X.400 or X.500 API functions that it uses.

You can declare a public object statically or you can create one dynamically at run time. You can only create a private object dynamically. You must consider this each time your application needs to pass an object as an argument to an X.400 API or an X.500 API function.

If data in your application can be declared statically, declare a public object and pass this to the interface. If data must be built dynamically, you can either build a public object or use the OM API to build a private object. The X.400 and X.500 APIs convert public objects into private objects before operating on them. It may therefore be more efficient to pass private objects.

3.1.1 Creating a Private Object

Use the Create function to create a new private object that is an instance of a class that you specify in the function call. The object class that you specify must be a concrete class; you cannot create an instance of an abstract class.

The X.400 and X.500 APIs include some concrete classes that are defined to return information to your application; for example, the Submission Results and Search-Result classes. You cannot create instances of these classes because they are intended to be returned by the API in question and never supplied to it.

You can choose to initialize the object with values specified in the class definition. For example, a directory service object of the Entry-Info-Selection class has three attributes, two of which can be initialized on creation:

- All-Attributes, which can be initialized to *true*
- Attribute-Selected, which is not initialized
- Info-Type, which can be initialized to *types-and-values*

To create the object with initial values, specify *true* for the Initialise argument of the Create function. Refer to Chapter 5 for a full description of the Create function, and to Chapter 4 for a list of the initial values supplied by the Create function.

The following example shows how to create a private object of Message Handling (MH) class Local Per-recipient NDR, with the Initialise argument set to *false*; the function does not initialize any of the object's attributes.

```
OM_private_object    ndr;
OM_workspace         workspace;
OM_return_code       result;

     result = om_create (MH_C_LOCAL_PER_RECIP_NDR,
                             /* class of object */
                         OM_FALSE,
                             /* do not initialise attributes */
                         workspace,
                             /* workspace in which object created */
                         &ndr);
                             /* created object */
```

The following example shows how to create a private object of the X.500 class Entry-Info-Selection, with the Initialise argument set to *true*; the function initializes two of the object's attributes.

```
OM_private_object    select_info;
OM_workspace         workspace;
OM_return_code       result;

     result = om_create (DS_C_ENTRY_INFO_SELECTION,
                             /* class of object */
                         OM_TRUE,
                             /* initialise attributes */
                         workspace,
                             /* workspace in which object created */
                         &select_info);
                             /* created object */
```

3.1.2 Creating a Public Object

To create a public object, declare a descriptor list in your application program. The following example shows the static declaration of an X.500 public object (`selection`), to be passed as an argument to the X.500 API DS Read function (`ds_read`). The object is an instance of the Entry-Info-Selection class.

```
#define ASTR "DS_A_COMMON_NAME"

static OM_descriptor selection[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
 {DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_FALSE},
 {DS_ATTRIBUTES_SELECTED, OM_S_OBJECT_IDENTIFIER_STRING, OM_STRING(ASTR)},
 {DS_INFO_TYPE, OM_S_ENUMERATION, DS_TYPES_AND_VALUES},
 OM_NULL_DESCRIPTOR
};
```

The example shows:

- The declaration of the class descriptor using the OM_OID_DESC macro.
- The declaration of descriptors to represent the object's attributes, DS_ALL_ATTRIBUTES, DS_ATTRIBUTES_SELECTED, and DS_INFO_TYPE.
- The OM syntaxes of the attribute values OM_S_BOOLEAN, OM_S_OBJECT_IDENTIFIER_STRING and OM_S_ENUMERATION.
- The attribute values, OM_FALSE, OM_STRING(ASTR) and DS_TYPES_AND_VALUES.
- The declaration of the null descriptor, OM_NULL_DESCRIPTOR.

OM_STRING is used to delimit the string that is the attribute value.

If you need to create public objects dynamically, you can do so using the Digital macros that are available in the <xom.h> header file. These allow you to set an empty descriptor to your required value, once you have declared a public descriptor or descriptor list. The following is an example:

```
/* Header files: */

   #include <xom.h>
   #include <xds.h>
   #include <xdsbdcp.h>

/* Declarations: */

   OM_EXPORT(DS_A_SURNAME)
   OM_EXPORT(DS_A_TITLE)

   OM_descriptor cpub_eis[5];

/* Assignment: */

   OMX_CLASS_DESC(       cpub_eis[0], DS_C_ENTRY_INFO_SELECTION);
   OMX_ATTR_TYPE_DESC(   cpub_eis[1], DS_ATTRIBUTES_SELECTED,DS_A_SURNAME);
   OMX_ATTR_TYPE_DESC(   cpub_eis[2], DS_ATTRIBUTES_SELECTED,DS_A_TITLE);
   OMX_ENUM_DESC(        cpub_eis[3], DS_INFO_TYPE, DS_TYPES_ONLY);
   OMX_OM_NULL_DESC(     cpub_eis[4]);
```

These macros are described in Chapter 7.

You can create a public object that is a copy of a private object using the Get function. This function is described in Section 3.3.

3.2 Modifying a Private Object

There are several ways of modifying a private object:

- Remove specified attribute values (see Section 3.2.1).
- Insert new attribute values or replace existing ones (see Section 3.2.2).
- Copy attribute values from a private object to an attribute (see Section 3.2.3).
- Write a segment of a string to an attribute (see Section 3.2.4).

3.2.1 Removing Attribute Values

If you wish to remove attribute values from a private object, you can do so using the Remove function. You can remove a single value (even if this is the only value in the attribute) or a range of values. If no attribute values remain after you use the Remove function, the attribute no longer exists (the function deletes the attribute).

When you use this function, you must specify the following:

- The private object whose attribute value, or values, you want to remove (in the Subject argument).
- The type of the attribute whose value, or values, you want to remove (in the Type argument).
- The position of the first value you want to remove (in the Initial Value argument).
- The position after the last value you want to remove (in the Limiting Value argument).

Attribute value positions start at zero. The first value in an attribute is therefore at position zero.

For a full description of the Remove function, refer to Chapter 5.

The following example shows a single value being removed from a private object of X.400 Message Handling class Message (`message`). `MH_T_LATEST_DELIVERY_TIME` is the attribute type whose value is removed. This attribute has one value. The function therefore removes the value and deletes the attribute.

```
OM_private_object    message;
OM_return_code       result;

     result = om_remove (message,
                         /* object from which to remove value */
                         MH_T_LATEST_DELIVERY_TIME,
                         /* attribute to remove */
                         0,1));
                         /* position of single value to remove */
```

The following example shows a range of values being removed from a private object of the X.500 class Entry-Information-Selection (select_info):

```
OM_private_object    select_info;
OM_return_code       result;

     result = om_remove (select_info,
                         /* object from which to remove value */
                         DS_ATTRIBUTES_SELECTED,
                         /* attribute to remove */
                         1,4));
                         /* range of values to remove */
```

DS_ATTRIBUTES_SELECTED is the attribute type whose values are removed. If this attribute has five values (positions 0 to 4), then this function removes the second, third and fourth values.

3.2.2 Putting Values into a Private Object

You can use the Put function to allocate values to attributes that have not been initialized, or to change attribute values. The Put function places copies of attribute values from one object (the source) into another object (the destination). The destination must be private; the source can be private or public.

Use the Included Types argument to specify the attribute type, or types, whose values you want to copy. If a specified attribute type is present in the source object, then all its values are copied to the attribute of the same type in the destination. If you do not specify any types, then the values of all attribute types in the source are copied to corresponding attributes in the destination.

You can specify that you want to replace values in the destination with source values, or that you want to insert the source values at particular points in the destination, leaving existing destination values unchanged. You specify the way that values are inserted into the destination using the Modification argument as follows:

- To insert values before any existing attribute values in the destination, specify *insert-at-beginning*.

- To insert values after any existing attribute values in the destination, specify *insert-at-end*.
- To remove existing values from the attribute in the destination and replace them with the values from the source, specify *replace-all*.
- To place values before the value at a specified position in the destination attribute, specify *insert-at-certain-point*.

  When you use this value, specify the position before which you want the values inserted as the Initial Value argument.
- To remove values at specified positions in the destination attribute and replace them with the values from the source attribute, specify *replace-certain-values*. The number of values removed from the destination does not have to equal those in the source. For example, you could remove five values and replace them with a single value, or remove a single value and replace it with two values.

  When you use this argument, specify the values to be replaced as follows:

  – The position of the first value to be replaced as the Initial Value argument

  – The position after the last value to be replaced as the Limiting value argument

The class of the source object is ignored; it has no effect on the destination object.

For a full description of the Put function, refer to Chapter 5.

The following example shows how to create an object of the X.500 class Entry-Mod (`modification`). The values from a public object of the same class, `pub_mod`, are put into the private object using Put.

```
#define ASTR "DS_A_COMMON_NAME"
#define STR1 "John Smith"
#define STR2 "Jack Smith"
#define STR3 "Jonathan Smith"

om_private_object    modification;
OM_return_code       result;
OM_workspace         workspace;
```

```
/* declare public object, pub_mod */
static OM_descriptor pub_mod[] = {
 OM_OID_DESC (OM_CLASS, DS_C_ENTRY_MOD),
 {DS_ATTRIBUTE_TYPE, OM_S_OBJECT_IDENTIFIER_STRING, OM_STRING(ASTR)},
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING(STR1)},
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING(STR2)},
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING(STR3)},
 {DS_MOD_TYPE, OM_S_ENUMERATION, DS_ADD_VALUES}
 OM_NULL_DESCRIPTOR
};

/* create private object, modification*/
     result = om_create (DS_C_ENTRY_MOD,
                             /* class of object */
                         OM_FALSE,
                             /* do not initialise attributes */
                         workspace,
                             /* workspace in which object created */
                         &modification);
                             /* created object */

/* put the values from the public object into the private object */

     result = om_put (modification,
                             /* destination object */
                      OM_REPLACE_ALL,
                             /* type of modification */
                      pub_mod,
                             /* source of values to be put */
                      0,0,0);
                             /* include all attributes, all positions */
```

After the call to the Put function, the private object `modification` contains all the attributes and values declared in the public object `pub_mod`.

3.2.3 Copying a String Attribute Value

Use the Copy Value function to copy a string attribute value from one private object to another. If the attribute to which you are copying has no existing value, this function inserts the copied value. Otherwise, the function overwrites the existing value.

Both values must be strings. The syntax of the copied value is the same as the syntax of the value from which the copy is made. For example, if the syntax of the attribute value from which you are copying is String(Graphic), the syntax of the attribute value to which you are copying is also String(Graphic).

When you use this function you must specify the following:

- The private object from which the string is copied (in the Source argument).

- The type of the attribute from which the string is copied (in the Source Type argument).
- The position in the attribute of the string that you want copied (in the Source Value Position argument).
- The private object into which the string is copied (in the Destination argument).
- The type of the attribute into which the string is copied (in the Destination Type argument).
- The position in the attribute into which the string is copied, replacing any existing value at that position (in the Destination Value argument).

For a full description of the Copy Value function, refer to Chapter 5.

The following example shows how to copy a string value between two objects of the X.400 Message Handling class Delivery Envelope (envelope1 and envelope2).

```
OM_private_object    envelope1,
                     envelope2;
OM_return_code       result;

     result = om_copy_value (envelope1,
                                  /* source object */
                             MH_T_CONTENT_IDENTIFIER,
                                  /* source attribute */
                             0,
                                  /* position of value in source attribute */
                             envelope2,
                                  /* destination object */
                             MH_T_CONTENT_IDENTIFIER,
                                  /* destination attribute */
                             0);
                                  /* position in destination attribute */
```

This call to Copy Value reads the string value at position 0 in the Content Identifier attribute of envelope1. This value is then copied to position 0 in the Content Identifier attribute of envelope2.

3.2.4 Writing a String Attribute Value in Segments

Use the Write function to write a segment of a string value into a private object. This function is useful for long string values.

You can call the function as many times as necessary to write a long string attribute value without having to place a copy of the whole value in memory. For example, you can use the Write function to write text into the body part of an X.400 message.

You can use the Write function to create new attribute values or to alter existing ones. You can insert segments into an existing attribute at any point, but any values already in the string beyond the insertion point are overwritten. The segment that you write therefore automatically becomes the last segment of the attribute value.

You do not have to use the Write function to put a long string value in an attribute. You can use the Put and Copy Value functions on long strings as well.

When you use this function, you must specify the following:

- The private object into which the segment is written (in the Subject argument).
- The type of the attribute into which the segment is written (in the Type argument).
- The position of the value in the attribute into which the segment is written (in the Value Position argument).
- The syntax of the value into which the segment is written, if the value does not already exist (in the Syntax argument).
- The offset, in octets, within the value, to start writing the segment (in the Starting Position argument).
- The string segment that is written to the value (in the Elements argument).

This function has an output argument, Next Position, which gives the offset (in octets) of the last segment written. You can use this value as the Starting Position argument of the next call to the Write function, thereby writing the string sequentially to an attribute value.

For a full description of the Write function, refer to Chapter 5.

The following example shows a string segment being written to an object of the X.400 Interpersonal Messaging class IA5 Text Body Part (body_part).

```
OM_return_code result;
OM_value_position position;

input_string = OM_STRING ("Text Body");
position = 0;

     result = om_write (body_part,
                             /* object containing string segment */
                        IM_TEXT,
                             /* attribute containing string segment */
                        0,
                             /* position of value to write to */
                        OM_S_IA5_STRING,
                             /* syntax of value */
                        &position,
                             /* starting position for current/next string */
                        input_string);
                             /* string segment to be written */
```

In this example the type of attribute being written to is IM_TEXT; the position of the value being written to is 0; the syntax of the value is OM_S_IA5_STRING.

3.3 Examining a Private Object

This section explains how to read the attribute values of a private object. Your application will need to do this so that it can process the results of certain function calls. A User Agent designed to use the X.400 API, for example, must examine the object returned from a call to the Start Delivery function in order to convey the information in it to the appropriate user or users. Similarly, an application that uses the X.500 API, must examine the object returned from a call to the Search function.

To examine the values of a private object, you must use the Get function. A private object is a nested data structure, and the Get function allows you to obtain data from any desired level in the structure. You can use the Get function in different ways:

- You can use the Get function to get all the data back in one go as a service-generated public object, that is, a descriptor list, and then write your own routines to work through the list. This method duplicates all the data that you ask for.

  If run-time speed is important and memory usage is not a consideration, then using this method avoids the overhead of multiple OM calls.

- You can use the Get function to work down the nested structure, getting data from one level at a time. This method does not duplicate data (from the private object to a public object) until you get to the values you want.

  Use this method if you wish to limit memory usage, or if the overhead of multiple OM calls is acceptable.

- You can use a combination of the above methods by working your way down the nested structure one level at a time until you get to the subobject you are interested in, and then retrieve all the data from that subobject in one go.

To work through a data structure one level at a time, use **exclusions** when calling the Get function. For example, you can exclude certain attribute types from the public copy. The exclusions you specify determine which subsets of the private object are copied into the public object.

You can combine exclusions by adding the values that denote individual exclusions.

When you use this function you must specify the following:

- The private object that you want a public copy of (in the Original argument).
- The exclusions to be applied (in the Exclusions argument). See Part II for details of the values of this argument.
- If you specified the value *exclude-all-but-these-types* in the Exclusions argument, then you must specify the attribute types that you want included in the copy (in the Included Types argument).
- Whether you want any string values translated into an implementation-defined local character set representation (in the Local Strings argument).
- If you specified the value *exclude-all-but-these-values* in the Exclusions argument, then you must specify the position within the original's attributes of the first value to be included in the copy (in the Initial Value argument).

  You must also specify the position within the original's attributes of the value after the last value to be included in the copy (in the Limiting Value argument).

Some attributes in the public copy may be subobjects. Descriptors representing these subobjects are returned by the call to the Get function, unless you specify *exclude-subobjects* in the Exclusions argument. If you specify *exclude-subobjects*, you only get a pointer (handle) to the private subobject. Therefore,

to examine the private subobjects, use the Get function again, passing the subobject as the Original argument to the function.

Note that if you specify *exclude-values*, the copy includes subobjects but without their values.

If the public copy contains an attribute with a syntax of long string, the function does not return a descriptor for that attribute. You must call the Read function to examine the attribute. Section 3.4 describes the Read function.

For a full description of the Get function, refer to Chapter 5.

The following example shows how to use the Get function to make a public copy of an object of the X.400 Message Handling class Local Per-recipient NDR (ndr) containing the attribute MH_T_TEMPORARY.

```
OM_private_object ndr;
OM_public_object  ndr_copy;
OM_value_position total_number;

OM_type
 temporary[] = {MH_T_TEMPORARY, OM_NO_MORE_TYPES};

     result = om_get (ndr,
                          /* object to be copied */
                      OM_EXCLUDE_ALL_BUT_THESE_TYPES,
                          /* include attributes of specified types */
                      temporary,
                          /* type to be included */
                      OM_FALSE,0,0,
                          /* no translation into local char set */
                      ndr_copy,
                          /* the copy */
                      &total_number);
                          /* number of attributes copied */
```

The example shows the use of the exclusion *exclude-all-but-these-types*. The public copy (ndr_copy) includes a descriptor representing the attribute MH_T_TEMPORARY (temporary), and its single value.

The following example shows how to use the Get function to work down a nested structure, getting one level at a time. The structure is a Result object returned from a call to the X.500 API DS Read function. Note that this code example uses the dsX_trace_object function, which is not available in the MAILbus 400 API.

```
/* declare an OM-type-list structure and variables to hold pointers to the
   entry, DS_object and RDNS subobjects:
   */

OM_integer        desc_count;
OM_object         read_result;
OM_type           included_types[2];
OM_public_object  spub_entry;
OM_public_object  spub_DS_object;
OM_public_object  spub_RDNS;

/* and set up the OM attributes you want to get first: */

included_types[0] = DS_ENTRY;
included_types[1] = OM_NO_MORE_TYPES;

/* now get only a pointer to the first subobject, the entry */

om_status = om_get(read_result,
       OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
       included_types, OM_FALSE, 0, OM_ALL_VALUES,
       &spub_entry, &desc_count);

/* the object spub_entry now contains only the
   OM-descriptor for an entry-information object */

dsX_trace_object(spub_entry);

/* Now use OM_get() again to extract the DN of the object */

included_types[0] = DS_OBJECT_NAME;
om_status = om_get(spub_entry->value.object.object,
       OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
       included_types, OM_FALSE, 0, OM_ALL_VALUES,
       &spub_DS_object, &desc_count);

dsX_trace_object(spub_DS_object);

/* Next, use OM_get() again to extract the RDNS */

included_types[0] = DS_RDNS;
om_status = om_get(spub_DS_object->value.object.object,
       OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
       included_types, OM_FALSE, 0, OM_ALL_VALUES,
       &spub_RDNS, &desc_count);

dsX_trace_object(spub_RDNS);

/* Now loop around each RDN, extract a pointer to the AVAS
   and then extract the attribute type and value
   */

   ...

/* When finished, remember to delete all the objects you have used */
```

3.4 Reading a String Attribute Value

Use the Read function to read a string in a private object, one segment at a time. This function enables you to read a specific segment of a long value without placing a copy of the entire value in memory. You specify which segment of the attribute value you want to read in your call to Read.

You must specify the following when you use this function:

- The private object from which the string value is read (in the Subject argument).
- The type of the attribute from which the string value is read (in the Type argument).
- The position in the attribute of the value that is read (in the Value Position argument).
- A Boolean value indicating whether the string is converted into the local character set representation (in the Local String argument).
- The position within the string value of the element that is read (in the First-Element-Position argument).
- A pointer to a variable of type OM_string into which the element is read (in the Elements argument).

The Read function has an output argument, Next Position, which gives the offset (in octets) within the string of the next segment to be read. You can use this value as the Starting Position argument of the next call to the Read function. Therefore, you can read a string sequentially from an attribute into the storage declared in the Elements argument.

For a full description of the Read function, refer to Chapter 5.

The following example shows how to read a string value from an object of the X.400 Interpersonal Messaging class IA5 Text Body Part (`body_part`). The Service will return 0 when there is no more text left to read.

```
OM_private_object   body_part;
OM_return code      result;
OM_string           message;
OM_string_length    offset;
char                read_buffer[1024];

message.length = 1024
message.elements = read_buffer;
offset = 0;
```

```
result = om_read (body_part,
                        /* object containing value to be read */
                  IM_TEXT,
                        /* attribute from which value is to be read */
                  0,
                        /* position of value to read from */
                  OM_FALSE,
                        /* no translation into local char set */
                  &offset,
                        /* string offset of segment to be read */
                  &message);
                        /* the string read from the value */
```

The type of the attribute read is IM_TEXT. The first element of the first value in this attribute is read into message.

3.5 Deleting an Object

You can use the Delete function to delete private objects or service-generated public objects when you no longer require them. This function releases all the resources that the Service assigned to the object.

3.5.1 Deleting a Service-Generated Public Object

When you use the Delete function to delete a service-generated public object, the function also deletes all the object's public subobjects. However, it does not delete its private subobjects.

For a full description of the Delete function, refer to Chapter 5.

The following example shows how to delete a service-generated public object of the X.500 class Entry-Info-Selection (info_select_copy). The object is a public copy of a private object of the Entry-Info-Selection class info_select.

```
OM_return_code      result;
OM_private_object   info_select;
OM_object           info_select_copy;
OM_value_position   total_number;
```

```
     result = om_get (info_select,
                              /* object to be copied */
                      OM_NO_EXCLUSIONS,
                              /* no exclusions */
                      NULL,
                              /* ignored because no exclusions specified */
                      OM_FALSE,0,0,
                              /* no translation into local char set */
                      &info_select_copy,
                              /* the copy */
                      &total_number);
                              /* number of attributes copied */

/* Examine info_select_copy public object using C programming language
constructs */

/* Delete object when finished with it */

     result = om_delete (info_select_copy);
                              /* the object to be deleted */
```

3.5.2 Deleting a Private Object

When you use the Delete function to delete a private object, the object and its private subobjects become inaccessible; that is, the handles of the object and of its subobjects, if any, become invalid.

If you use the function to delete a private object that is a subobject of another private object, then the subobject becomes inaccessible by its own handle, but is still accessible if you use the Get function on the private object that contains it. A private object is only deleted when all handles to it are invalid.

The following example shows how to delete a private object of the X.400 Message Handling class Local NDR (ndr), created using the Create function.

```
OM_return_code      result;
OM_workspace        workspace;
OM_private_object   ndr;

     result = om_create (MH_C_LOCAL_NDR,
                              /* class of object */
                         OM_TRUE,
                              /* initialise attributes */
                         workspace,
                              /* workspace */
                         &ndr);
                              /* created object */

/* Manipulate ndr private object using OM functions, and use it as
input to the X.400 API Message Access (MA) Finish-Delivery function,
ma_finish_delivery */
```

```
/* Delete object when finished with it */
     result = om_delete (ndr);
                           /* the object to be deleted */
```

3.6 Copying an Object

The Copy function allows you to create an exact copy of a private object and its subobjects. The service places the copy in a workspace that you specify when you call the Copy function. The copy is independent of the original and so when you modify the copy, the original is not affected.

Specify the following when you call this function:

- The private object to be copied.
- The workspace into which the copy is to be placed.
- A pointer to a private object, which will contain the copy.

For a full description of the Copy function, refer to Chapter 5.

The following example shows how to copy an object of the X.400 Message Handling class Local NDR (ndr). The copy is ndr_copy.

```
OM_return_code      result;
OM_private_object   ndr,
                    ndr_copy;
OM_workspace        workspace;

     result = om_copy (ndr,
                           /* object to be copied */
                       workspace,
                           /* workspace in which to create copy */
                       &ndr_copy);
                           /* the copy */
```

3.7 Determining the Class of an Object

The Instance function enables you to find out whether an object is an instance of a specified class. You can use this function to determine the class of private or service-generated public objects.

For example, suppose you want to check that Object A is an instance of the X.500 class, Attribute-List. Object A is an instance of the Entry-Info class, which is a subclass of Attribute-List. Examining the Class attribute of Object A gives the value Entry-Info. However, using the Instance function with Object A's Handle and the identifier of the Attribute-List class as arguments confirms that Object A is also an instance of the Attribute-List class.

Specify the following when you call this function:

- The name of the object whose class you wish to check (in the Subject argument).
- The class that you are testing for (in the Class argument).
- A pointer to a variable of type Boolean that will contain the result of the query.

After the application has called the function, the Boolean variable contains the value *true* if the object is an instance of the given class (or one of its subclasses), and *false* if it is not.

For a full description of the Instance function, refer to Chapter 5.

The following example shows how to check a private object, an_object, to see if it is an instance of the X.400 Message Handling class Submitted Message RD. inst is the return argument. The function returns *true* in the parameter if the object is an instance of the Submitted Message RD class, or its subclass Message RD.

```
OM_private_object   an_object;
OM_return_code      result;
OM_boolean          inst;

     result = om_instance (an_object,
                           /* object to be checked */
                           MH_C_SUBMITTED_MESSAGE_RD,
                           /* class to be checked against */
                           &inst);
                           /* result */

     if (inst == OM_TRUE)
              /* the object is an instance of specified class */

     else
              /* the object is not an instance of specified class */
```

3.8 Encoding and Decoding Private Objects

The OM API provides two functions, Encode and Decode, which respectively enable you to encode private objects, and to decode encoded objects back into private objects. The current version only supports the encoding of objects according to ASN.1 Basic Encoding Rules (BER).

The advantage of encoding data is that it becomes completely transportable; the data can have only one meaning, regardless of the hardware platform.

3.8.1 Encoding

The Encode function enables you to create a new private object that is the encoded version of an existing private object. Because the encoding can only be done according to the Basic Encoding Rules, you must specify OM_BER in the function call.

For a full description of the Encode function, refer to Chapter 5.

The following example shows the encoding of an object of the X.400 Message Handling class Report (encodable_object). The object is encoded according to the rules OM_BER, and the encoded object is encoding.

```
OM_return_code      result;
OM_private_object   encodable_object,
                    encoding;

     result = om_encode (encodable_object,
                         /* object to be encoded */
                         OM_BER,
                         /* encoding rules */
                         &encoding);
                         /* encoded object */
```

3.8.2 Decoding

The Decode function creates a new private object that is a decoded version of an existing encoded private object. The copy is independent and any alterations you make to the original do not affect the copy.

For a full description of the Decode function, refer to Chapter 5.

The following example shows the decoding of the object encoded in the code example from Section 3.8.1. The encoded object is encoding, and the decoded object is decoded_object.

```
OM_return_code      result;
OM_private_object   encoding,
                    decoded_object;

     result = om_decode (encoding,
                         /* object to be decoded */
                         &decoded_object);
                         /* decoded object */
```

The decoded object is an instance of the X.400 Message Handling class Report, the same as the object that was previously encoded.

Part II
Reference

This part gives reference information for the OM API. It contains five chapters and three appendixes:

- Chapter 4: this describes the Object Management Package.
- Chapter 5: this describes the OM API functions, for example the Get and Put functions.
- Chapter 6: this describes the syntaxes that OM attribute values can have.
- Chapter 7: this describes the data types defined for OM.
- Chapter 8: this describes the OM API header files.
- Appendix A: this lists the declarations that define the symbolic constants for the C interface of the OM API.
- Appendix B: this lists the characters that are allowed in string types Printable, Numeric and IA5.
- Appendix C: this lists and explains the return values used by the functions in the OM API.

4
Object Management Package

This chapter describes the Object Management (OM) package.

4.1 OM Package Object Identifier

An object identifier is assigned to the OM package. In Abstract Syntax Notation One (ASN.1), it is specified as:

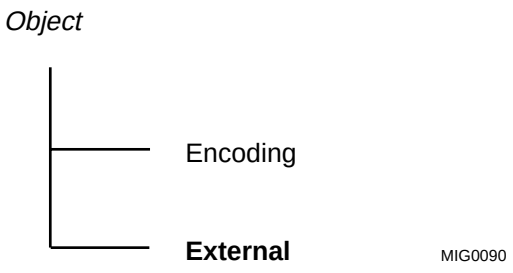
{joint-iso-ccitt mhs-motis(6) group(6) white(1) api(2) om(4)}

The OM API header file includes a symbol definition for this object identifier. The symbol is OM_OM.

4.2 Class Hierarchy

Figure 4–1 shows the Object Management class hierarchy. The italic font used for the Object class indicates that it is an abstract class. The bold font used for the External class indicates that it is a class to which `om_encode` applies. `om_create` applies to both concrete classes. (The OM functions are described in Chapter 5.)

Figure 4–1 Class Hierarchy of the OM Classes



Section 4.3 contains definitions of these classes.

4.3 Class Definitions

This section describes the classes in the OM package. The attributes belonging to each class are shown in tables with the following columns:

- OM Attribute

  Lists the attributes specific to the class.

- Value Syntax

  Gives the syntax for each attribute.

- Value Length

  Indicates constraints on the number of bits, octets, or characters in each value that is a string.

- Number of Values

  Indicates constraints on the number of values (and therefore whether the value is optional or mandatory).

- Initial Value

  Shows the values that `om_create` supplies when you initialize the application.

Beneath each table is a list describing the attributes in the table.

4.3.1 Object

OM_C_OBJECT

The Object class represents information objects. This abstract class has no superclass, and all other classes in the OM Package are its subclasses.

The Object class has the attribute shown in the following table.

OM Attribute	Value Syntax	Value Length	Number of Values	Initial Value
Class	String(Object Identifier)	-	1	-

Class (OM_CLASS)
Specifies the class of the object.

4.3.2 Encoding

OM_C_ENCODING

An instance of the Encoding class is an object in a form suitable for passing between workspaces, for transport across a network, or for storage in a file. The form may also be suitable for presentation to an intermediate service provider, for example a Directory Service or a Message Transfer System that would not otherwise recognize the object.

This class has the attributes of its superclass plus those listed in the following table.

OM Attribute	Value Syntax	Value Length	Number of Values	Initial Value
Object Class	String(Object Identifier)	-	1	-
Object Encoding	String(Encoding)	-	1	-
Rules	String(Object Identifier)	-	1	BER

Object Class (OM_OBJECT_CLASS)
Identifies the class of the object that the Object Encoding attribute encodes. The class should be a concrete class.

Object Encoding (OM_OBJECT_ENCODING)
The encoding.

Rules (OM_RULES)
Identifies the set of rules that were followed to produce the Object Encoding attribute. The current version of the OM API only supports the ASN.1 Basic Encoding Rules (BER), the symbol for which is OM_BER.

NOTE

An object of the Encoding class must appear as a value whose syntax is Object(Encoding), even if the encoded object belongs to some other class.

4.3.3 External

OM_C_EXTERNAL

An instance of the External class describes a data value and identifies its data type. This class corresponds to the ASN.1 class External.

This class has the attributes of its superclass plus those listed in the following table.

OM Attribute	Value Syntax	Value Length	Number of Values	Initial Value
Arbitrary Encoding	String(Bit)	-	0-1[1]	-
ASN1 Encoding	String(Encoding)	-	0-1[1]	-
Data Value Descriptor	String(Object Descriptor)	-	0-1	-
Direct Reference	String(Object Identifier)	-	0-1	-
Indirect Reference	Integer	-	0-1	-
Octet Aligned Encoding	String(Octet)	-	0-1[1]	-

[1]Exactly one of these three attributes must be present in an instance of the External class.

Arbitrary Encoding (OM_ARBITRARY_ENCODING)
Represents the data value as a bit string.

ASN1 Encoding (OM_ASN1_ENCODING)
The data value, present only if the data type is an ASN.1 type.

Data Value Descriptor (OM_DATA_VALUE_DESCRIPTOR)
A description of the data value.

Direct Reference (OM_DIRECT_REFERENCE)
A direct reference to the data type.

Indirect Reference (OM_INDIRECT_REFERENCE)
An indirect reference to the data type.

Octet Aligned Encoding (OM_OCTET_ALIGNED_ENCODING)
Represents the data value as an octet string.

4.4 C Naming Conventions

The OM API supports only the C programming language. How a C identifier is derived depends upon the type of element, as described in Table 4–1. The C identifier has the prefix listed in the column headed Prefix. The remainder of the identifier is in the case listed in the column headed Case.

Table 4–1 C Naming Conventions

Element Type	Prefix	Case
Data type	om_	Lower
Data value	OM_	Upper
Data value (class 1)	OM_C_	Upper
Data value (syntax)	OM_S_	Upper
Data value component (structure member)	–	Lower
Function	om_	Lower
Function argument	–	Lower
Function result	–	Lower
Macro	OM_	Upper
Reserved for use by implementors	OMP	Any
Reserved for use by implementors	omP	Any
Reserved for proprietary extension	omX	Any
Reserved for proprietary extension	OMX	Any

5

OSI-Abstract-Data Manipulation Functions

This chapter describes the OM API functions. The functions are listed in alphabetical order.

Each function description includes a list of return values. For an explanation of these values, refer to Appendix C.

om_copy

Creates a copy of an existing private object.

Syntax

OM_return_code om_copy (original, workspace, copy)

Argument	Data Type	Access
original	OM_private object	read
workspace	OM_workspace	read
copy	OM_private_object	write
return_code	OM_return_code	

C Binding

```
OM_return_code om_copy (original, workspace, copy)

OM_private_object   original,
OM_workspace        workspace,
OM_private_object   *copy
```

Arguments

Original
The original private object.

Workspace
The workspace in which the Service creates the copy. The workspace that the Client specifies in this argument must be one that is associated with a package containing the class of the original object.

Copy
The copy of the original object. The Service returns this argument if the Return Code of the function is OM_SUCCESS.

Description

This function creates a new private object, the copy, which is an exact but independent copy of an existing private object, the original. The function also copies the original's subobjects, if it has any.

The Client can specify a workspace in which the Service should place the copy. If the Client does not do so, the Service places the copy in the original's workspace.

Return Values

OM_SUCCESS
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_CLASS
OM_NO_SUCH_OBJECT
OM_NO_SUCH_WORKSPACE
OM_NOT_PRIVATE
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR
OM_TOO_MANY_VALUES

om_copy_value

Copies a value (string) from a private object and places it in another private object.

Syntax

OM_return_code om_copy_value (source, source_type, source_value_position, destination, destination_type, destination_value_position)

Argument	Data Type	Access
source	OM_private_object	read
source_type	OM_type	read
source_value_position	OM_value_position	read
destination	OM_private_object	read
destination_type	OM_type	read
destination_value_position	OM_value_position	read
return_code	OM_return_code	

C Binding

OM_return_code om_copy_value (source, source_type, source_value_position, destination, destination_type, destination_value_position)

OM_private_object source,
OM_type source_type,
OM_value_position source_value_position,
OM_private_object destination,
OM_type destination_type,
OM_value_position destination_value_position

Arguments

Source
The object from which you want to copy the value.

Source Type
The type of the attribute value from which you want to copy the value.

Source Value Position
The position within the attribute of the value to be copied.

Destination
The object to which you want to copy the value.

Destination Type
The type of the attribute to which you want to copy the value.

Destination Value Position
The position within the destination attribute at which you want to place the copied value. If the value of this argument exceeds the number of values in the Destination attribute, then it is taken to be equal to that number.

Description

This function either replaces, or fills in for the first time, an attribute value in the destination object with a copy of an attribute value from the source object. The source value should be a string. The copy has the same syntax as the source value. See Chapter 7 for more information on the String data type.

Return Values

OM_SUCCESS
OM_FUNCTION_DECLINED
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_OBJECT
OM_NO_SUCH_TYPE
OM_NOT_PRESENT
OM_NOT_PRIVATE
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR
OM_WRONG_VALUE_LENGTH
OM_WRONG_VALUE_SYNTAX
OM_WRONG_VALUE_TYPE

om_create

Creates a new private object that is an instance of a particular class.

Syntax

OM_return_code om_create (class, initialise, workspace, object)

Argument	Data Type	Access
class	OM_object_identifier	read
initialise	OM_boolean	read
workspace	OM_workspace	read
object	OM_private_object	write
return_code	OM_return_code	

C Binding

```
OM_return_code  om_create  (class, initialise, workspace, object)

OM_object_identifier    class,
OM_boolean              initialise,
OM_workspace            workspace,
OM_private_object       *object
```

Arguments

Class

The class of the object you are creating. It must be a concrete class.

Initialise

If you set this argument to OM_TRUE, the object that you create has some of its attributes initialised. These are the attributes for which initial values are specified in the class definition table. You can find these class definition tables in the documentation for the X.400 and X.500 APIs.

If you set this argument to OM_FALSE, the object you create has only its Class attribute initialised.

Workspace

The workspace in which the Service should create the object. The class you specify for the object must be in a package that you have already associated

with this workspace. Chapter 2 explains how to associate a package with a workspace.

Object
This is the created object. The Service returns this argument if the Return Code of the function is OM_SUCCESS.

Description

This function creates a private object in the workspace that you specify.

You can add new values and replace or remove existing values, any time after the object has been created. In this way, you can create any possible instance of the object's class.

Return Values

OM_SUCCESS
OM_FUNCTION_DECLINED
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_CLASS
OM_NO_SUCH_WORKSPACE
OM_NOT_CONCRETE
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR

om_decode

Creates a new private object that decodes an existing ASN.1 private object.

Syntax

OM_return_code om_decode (encoding, original)

Argument	Data Type	Access
encoding	OM_private_object	read
original	OM_private_object	write
return_code	OM_return_code	

C Binding

```
OM_return_code om_decode (encoding, original)

OM_private_object      encoding,
OM_private_object     *original
```

Arguments

Encoding
The encoded object that you want to decode. It must be an instance of the Encoding class.

Original
An object that is the decoded version of the encoding. The Service creates this object in the workspace in which the encoding is located. The Service returns this argument if the Return Code of the function is OM_SUCCESS.

Description

This function creates a new private object by decoding the ASN.1 of the original object.

In the Encoding argument, you specify the class of the existing object and the rules used to encode it. In the current version of the OM API, you must specify ASN.1 BER.

Return Values

OM_SUCCESS
OM_ENCODING_INVALID
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_CLASS
OM_NO_SUCH_OBJECT
OM_NO_SUCH_RULES
OM_NOT_AN_ENCODING
OM_NOT_PRIVATE
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR
OM_TOO_MANY_VALUES
OM_WRONG_VALUE_LENGTH
OM_WRONG_VALUE_MAKEUP
OM_WRONG_VALUE_NUMBER
OM_WRONG_VALUE_SYNTAX
OM_WRONG_VALUE_TYPE

om_delete

Deletes a service-generated public object or makes a private object inaccessible.

Syntax

OM_return_code om_delete (subject)

Argument	Data Type	Access
subject	OM_object	read
return_code	OM_return_code	

C Binding

OM_return_code om_delete (subject)

OM_object subject

Arguments

Subject
The object that you want the Service to delete. It must be a service-generated public object or a private object. If the object that you specify is a client-generated public object, the function returns an error status.

Description

This function deletes a service-generated public object, or makes a private object inaccessible.

When you apply this function to a service-generated public object, the function deletes the object and releases the resources associated with it. The resources include the space occupied by descriptors and attribute values. The function also deletes all public subobjects of the subject. This function does not delete private subobjects.

When you apply this function to a private object, the function makes the object inaccessible by making its Handle invalid. The function also makes invalid the Handles of any private subobjects of the subject. Note that the effect of using an object's Handle once it has been made invalid is undefined.

Return Values

OM_SUCCESS
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_OBJECT
OM_NO_SUCH_SYNTAX
OM_NO_SUCH_TYPE
OM_NOT_THE_SERVICES
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR

om_encode

Creates a new private object that encodes an existing private object.

Syntax

OM_return_code om_encode (original, rules, encoding)

Argument	Data Type	Access
original	OM_private_object	read
rules	OM_object_identifier	read
encoding	OM_private_object	write
return_code	OM_return_code	

C Binding

OM_return_code om_encode (original, rules, encoding)

```
OM_private_object       original,
OM_object_identifier    rules,
OM_private_object       *encoding
```

Arguments

Original
The object you want to encode.

Rules
The set of rules that the Service must follow to produce an encoding. In this version of the OM API, you can only specify ASN.1 BER.

Encoding
An object that is the encoded version of the original. The Service creates this object in the workspace in which the original is located. The Service returns this argument if the Return Code of the function is OM_SUCCESS. The returned object is an instance of the Encoding class.

Description

This function creates a new private object, the encoding, which exactly and independently encodes an existing private object, the original.

When you apply this function to a private object, the function uses the encoding rules you specify to create a new private object. The new encoded private object is independent of the original private object.

Return Values

OM_SUCCESS
OM_FUNCTION_DECLINED
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_OBJECT
OM_NO_SUCH_RULES
OM_NOT_PRIVATE
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR

om_get

Creates a new public object that is a copy of the whole or part of a private object.

Syntax

```
OM_return_code  om_get  (original, exclusions, included_types, local_strings, initial_value,
                         limiting_value, copy, total_number)
```

Argument	Data Type	Access
original	OM_private_object	read
exclusions	OM_exclusions	read
included_types	OM_type_list	read
local_strings	OM_boolean	read
initial_value	OM_value_position	read
limiting_value	OM_value_position	read
copy	OM_public_object	write
total_number	OM_value_position	write
return_code	OM_return_code	

C Binding

```
OM_return_code  om_get  (original, exclusions, included_types, local_strings, initial_value,
                         limiting_value, copy, total_number)

OM_private_object       original,
OM_exclusions           exclusions,
OM_type_list            included_types,
OM_boolean              local_strings,
OM_value_position       initial_value,
OM_value_position       limiting_value,
OM_public_object        *copy,
OM_value_position       *total_number
```

Arguments

Original

The private object, all or part of which you want to copy.

Exclusions

A list of zero or more values, each of which reduces the copy to a portion of the original. The exclusions apply to the attributes of the original object, but not to the attributes of its subobjects. This argument has one or more of the following values:

- OM_EXCLUDE_ALL_BUT_THESE_TYPES

  The copy includes descriptors of attributes of specified types only.

- OM_EXCLUDE_MULTIPLES

  The copy includes a single descriptor for each attribute having two or more values, instead of one descriptor for each value. Each such descriptor contains no attribute value, and the No-Value bit of the syntax component is set. If the attribute has values of two or more syntaxes, the descriptor identifies one of those syntaxes. Which syntax it identifies is unspecified.

- OM_EXCLUDE_ALL_BUT_THESE_VALUES

  The copy includes descriptors encompassing only values at specified positions within an attribute.

- OM_EXCLUDE_VALUES

  The copy includes a single descriptor for each attribute value, but the descriptor does not contain the value, and the No-Value bit of the syntax component is set.

- OM_EXCLUDE_SUBOBJECTS

  The copy includes a descriptor for each value which has a syntax of object. Each descriptor contains an object Handle for the original private subobject, instead of a public copy of the original private subobject. The Handle makes the private subobject accessible for use in OM function calls.

- OM_EXCLUDE_DESCRIPTORS

  The function does not return any descriptors, nor does it return a value in the Copy argument. The value of the Total Number argument gives the number of descriptors which would have otherwise been returned.

- OM_NO_EXCLUSIONS

  The copy contains descriptors and values for all attributes in the original.

If you specify multiple exclusions, the Service applies the exclusions in the order in which they occur in the above list. If a portion of the object disappears after the Service applies an exclusion, the Service applies no further exclusions to that portion.

Included Types
The types of attributes that are to be included in the copy, provided they appear in the original. This argument must be present if you select the OM_EXCLUDE_ALL_BUT_THESE_TYPES exclusion, but must otherwise be set to null.

Local Strings
If you set this argument to OM_TRUE, you indicate to the Service that it must translate all String(*) values included in the Copy into the local character set representation. This translation may cause the loss of some information.

Initial Value
The position within each attribute of the first value to be included in the copy. This argument must be present if you select the OM_EXCLUDE_ALL_BUT_ THESE_VALUES exclusion, but must otherwise be set to null.

If the value of Initial Value is OM_ALL_VALUES, or if it exceeds the number of values present in an attribute, the Service takes Initial Value to be equal to the number of values present in the attribute.

Limiting Value
The position in each attribute that is one element beyond the position of the last value included in the copy. This argument must be present if you select the OM_EXCLUDE_ALL_BUT_THESE_VALUES exclusion, but must otherwise be set to null.

If the value of Limiting Value is less than that of Initial Value, the Service does not put any values in the copy.

If the value of Limiting Value is OM_ALL_VALUES, or if it exceeds the number of values present in an attribute, then the Service takes Limiting Value to be equal to the number of values present in the attribute.

Copy
An exact but independent copy of the original. The Service returns this argument if both the following conditions are true:

- The Return Code of the function is OM_SUCCESS
- You do not specify the OM_EXCLUDE_DESCRIPTORS exclusion

You do not have to allocate any space to the copy. If you alter any portion of this space, you may affect the behaviour of the Service.

Total Number
The number of attribute descriptors in the copy. This does not include descriptors in any subobjects of the copy. If you specify the OM_EXCLUDE_ DESCRIPTORS exclusion, then there is no copy. Therefore, the value in Total Number is the number of descriptors that the Service would return if you did not specify OM_EXCLUDE_DESCRIPTORS (applying any other exclusions that you specified).

NOTE

Total Number excludes the special descriptor that signals the end of a public object, OM_NULL_DESCRIPTOR.

Description

This function creates a new public object, the copy, which is an exact but independent copy of an existing private object, the original.

When using this function, you can request certain exclusions, each of which reduces the copy to a portion of the original.

When this function is used with the X.400 API, one exclusion is requested implicitly. For each attribute value in the original that is a string exceeding 1024 bytes in length, the copy includes a descriptor that omits the elements, but specifies the length of the string. In this case, the following applies:

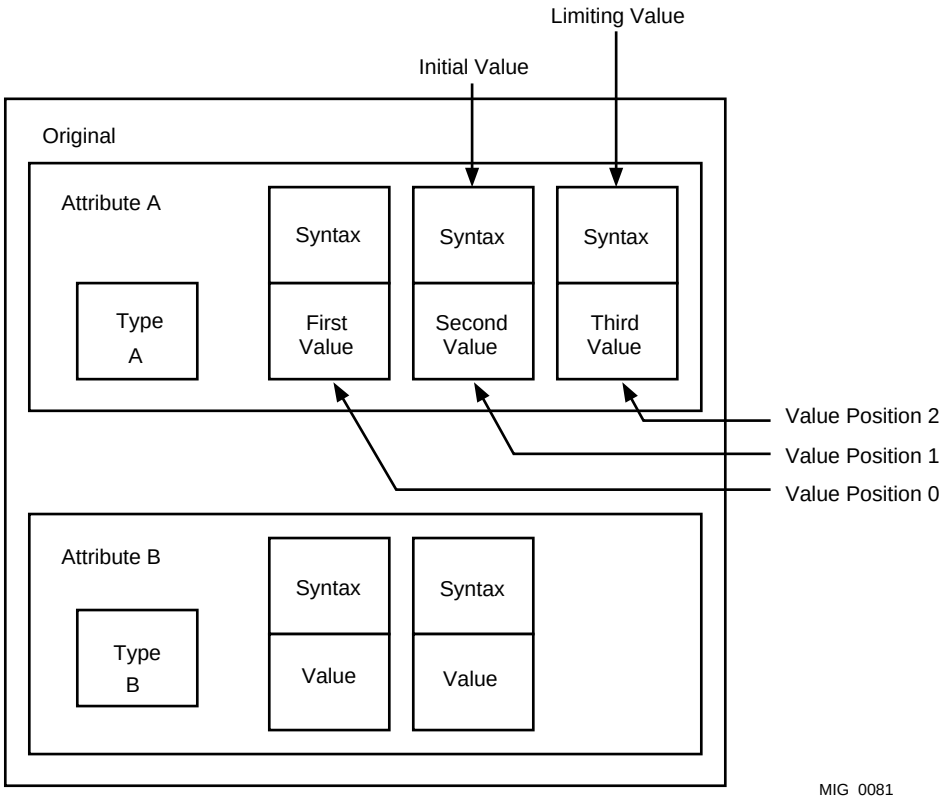
- The syntax of the descriptor has its Long-String bit set.
- The value of the descriptor is a string whose elements component is set to OM_ELEMENTS_UNSPECIFIED, but whose length component *does* specify the correct length.

Note that if you are using the OM API with the X.400 API, you can read long strings using `om_read`.

When used with the X.500 API, there is no limit to the length of string returned by `om_get`, and so this exclusion is not requested implicitly.

Figure 5–1 shows the original object, with Initial Value and Limiting Value labelled.

Figure 5–1 Original Object

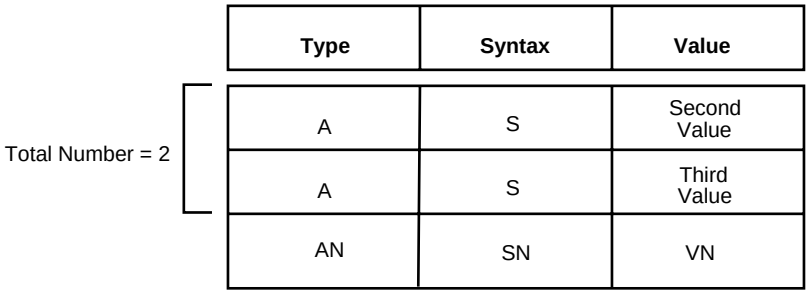


In this example, the call to `om_get` includes the following argument values:

- Exclusions has the value `OM_EXCLUDE_ALL_BUT_THESE_VALUES + OM_EXCLUDE_ALL_BUT_THESE_TYPES`
- Included Types lists Type A.
- Local Strings has the value OM_FALSE.
- Initial Value has the value 1, and Limiting Value the value 2.

Figure 5–2 shows the Copy and Total Number after a call to om_get.

Figure 5–2 Public Object



Key:

AN = OM_NO_MORE_TYPES
SN = OM_S_NO_MORE_SYNTAXES
VN = OM_LENGTH_UNSPECIFIED, OM_ELEMENTS_UNSPECIFIED MIG0084

Return Values

OM_SUCCESS
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_EXCLUSION
OM_NO_SUCH_OBJECT
OM_NO_SUCH_TYPE
OM_NOT_PRIVATE
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR
OM_WRONG_VALUE_SYNTAX
OM_WRONG_VALUE_TYPE

om_instance

Determines whether an object is an instance of a particular class or of one of its subclasses.

Syntax

OM_return_code om_instance (subject, class, instance)

Argument	Data Type	Access
subject	OM_object	read
class	OM_object_identifier	read
instance	OM_boolean	write
return_code	OM_return_code	

C Binding

```
OM_return_code  om_instance  (subject, class, instance)

OM_object             subject,
OM_object_identifier  class,
OM_boolean           *instance
```

Arguments

Subject
The object whose class you want to verify.

Class
The class against which you want to verify the subject.

Instance
The Service sets this argument to OM_TRUE if the subject is an instance of the class you specified, and OM_FALSE if the subject is not. The Service returns this argument if the Return Code of the function is OM_SUCCESS.

Description

This function enables you to determine whether an object is an instance of a specified class or of any of the subclasses of that class.

Note that it is possible to determine an object's class by using programming constructs to inspect the object, if it is public, or by using `om_get`, if it is private. The advantage of the Instance function is that it indicates whether the object is an instance of the specified class, even when it is also a subclass of the specified class.

Return Values

OM_SUCCESS
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_CLASS
OM_NO_SUCH_OBJECT
OM_NO_SUCH_SYNTAX
OM_NOT_THE_SERVICES
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR

om_put

Places copies of the attribute values of a private or public object into a private object.

Syntax

```
OM_return_code  om_put  (destination, modification, source, included_types, initial_value,
                         limiting_value)
```

Argument	Data Type	Access
destination	OM_private_object	read
modification	OM_modification	read
source	OM_object	read
included_types	OM_type_list	read
initial_value	OM_value_position	read
limiting_value	OM_value_position	read
return_code	OM_return_code	

C Binding

```
OM_return_code  om_put (destination, modification, source, included_types, initial_value,
                        limiting_value)

OM_private_object   destination,
OM_modification     modification,
OM_object           source,
OM_type_list        included_types,
OM_value_position   initial_value,
OM_value_position   limiting_value
```

Arguments

Destination

The object into which you want to put attribute values. This function does not affect the class of the destination.

Modification

A list of modifications to the attributes selected for copying. The modifications you request determine how the function modifies the destination object with the attributes, that is, where it puts them.

The Modification argument can have one of the following values:

- OM_INSERT_AT_BEGINNING

  The Service inserts the source values before all existing destination values. This does not affect the existing destination values.

- OM_INSERT_AT_CERTAIN_POINT

  The Service inserts the source values before the value at a specified position in the destination attribute. This does not affect the existing destination values.

- OM_INSERT_AT_END

  The Service inserts the source values after all existing destination values. This does not affect the existing destination values.

- OM_REPLACE_ALL

  The Service replaces any destination values with the source values, and discards the original destination values.

- OM_REPLACE_CERTAIN_VALUES

  The Service replaces the values at specified positions in the destination attribute with values from the source. The Service discards the original destination attribute values at those positions.

Source

The object from which you want to copy attribute values. This function ignores the class of the source.

Included Types

The types of attributes that should be copied to the destination, if they appear in the source. If you do not specify a value for this argument, the Service copies all attributes from the source to the destination.

Initial Value

You need to pass a value for this argument when you select the OM_INSERT_AT_CERTAIN_POINT modification or the OM_REPLACE_CERTAIN_VALUES modification. The following table shows what the Initial Value argument represents in each of these cases.

Modification	Meaning of the Initial Value Argument
insert-at-certain-points	The position within each destination attribute at which the Service should insert source values
replace-certain-values	The first value that the Service should replace

If you give this argument a value that is greater than the number of values present in a destination attribute, or if you use the value OM_ALL_VALUES, the Service takes Initial Value to be equal to the number of values present in the destination attribute.

Limiting Value

You need to pass a value for this argument when you select the OM_REPLACE_CERTAIN_VALUES modification. The argument specifies the position within each destination attribute that is one beyond that of the last value to be replaced. The value of Limiting Value must be greater than that of Initial Value.

If you give this argument a value that is greater than the number of values present in a destination attribute, or if you use the value OM_ALL_VALUES, the Service takes Limiting Value to be equal to the number of values present in the destination attribute.

Description

This function places in one private object, the destination, copies of the attribute values of another object, the source. The source can be a public or private object.

You must specify that the Service does one of the following:

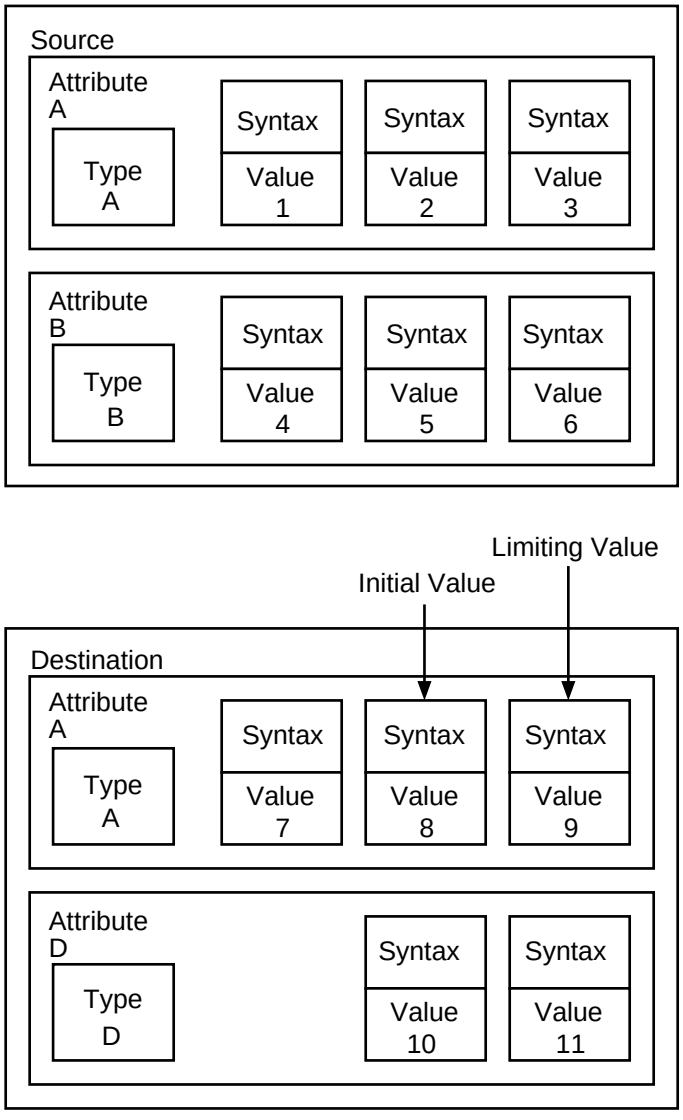
- Replace all the values in the destination with values from the source.
- Replace specified values in the destination with values from the source.
- Insert values from the source in a particular position in the destination.

Only use `om_put` to copy attributes from the source that occur in the definition of the class to which the destination belongs.

The Service first converts all string values that are in the local representation into the non-local representation for that syntax.

Figure 5–3 shows an example of a source object and a destination object before a call to `om_put`, with Initial Value and Limiting Value labelled.

Figure 5–3 Source and Destination Objects Before Copying Attribute Values



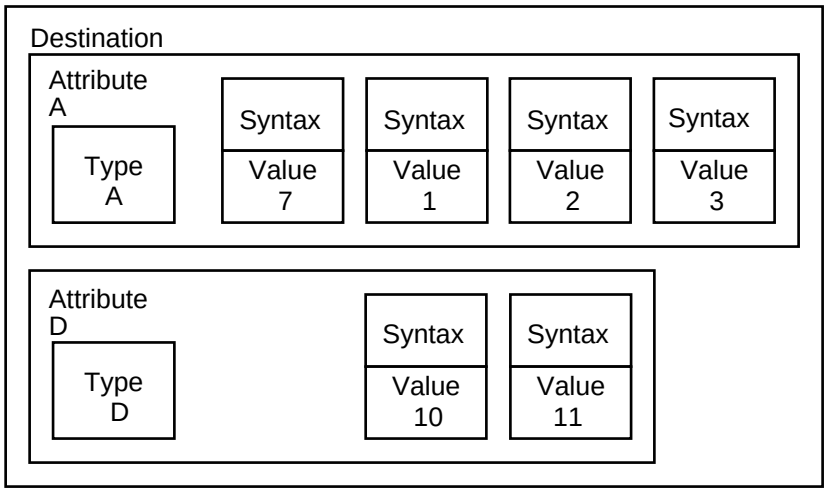
In this example, the call to `om_put` includes the following argument values:

- Modification has the value OM_REPLACE_CERTAIN_VALUES

- Included Types lists Type A and Type C

Figure 5–4 shows the destination object after the call to Put.

Figure 5–4 Destination Object After Copying Attribute Values



MIG 0078

There is no attribute of type C in the source object. The destination object therefore contains no attributes of this type, even though type C is specified in the Included Types argument.

The destination object contains an attribute, D, which is not affected by `om_put`.

Return Values

OM_SUCCESS
OM_FUNCTION_DECLINED
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_CLASS
OM_NO_SUCH_MODIFICATION
OM_NO_SUCH_OBJECT
OM_NO_SUCH_SYNTAX

OM_NO_SUCH_TYPE
OM_NOT_CONCRETE
OM_NOT_PRESENT
OM_NOT_PRIVATE
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR
OM_TOO_MANY_VALUES
OM_VALUES_NOT_ADJACENT
OM_WRONG_VALUE_LENGTH
OM_WRONG_VALUE_MAKEUP
OM_WRONG_VALUE_NUMBER
OM_WRONG_VALUE_POSITION
OM_WRONG_VALUE_SYNTAX
OM_WRONG_VALUE_TYPE

om_read

Reads a segment of a string from a private object.

Syntax

OM_return_code om_read (subject, type, value_position, local_string, string_offset, elements)

Argument	Data Type	Access
subject	OM_private_object	read
type	OM_type	read
value_position	OM_value_position	read
local_string	OM_boolean	read
string_offset	OM_string_length	read-write
elements	OM_string	write
return_code	OM_return_code	

C Binding

OM_return_code om_read (subject, type, value_position, local_string, string_offset, elements)

```
OM_private_object     subject,
OM_type               type,
OM_value_position     value_position,
OM_boolean            local_string,
OM_string_length     *string_offset,
OM_string            *elements
```

Arguments

Subject
The private object from which you want to read the segment.

Type
The type of the attribute containing the value that you want to read.

Value Position
The position in a multi-valued attribute of the value that you want to read.

Local String

If you set this argument to OM_TRUE, the Service translates the attribute segment into the local character set. This translation may result in the loss of some information.

String Offset

If provided by the Client, this argument denotes the position within the attribute value of the first element that you want to read. If you give this argument a value that exceeds the number of elements present in the attribute value, the Service takes the argument to be equal to the number of elements present in the attribute value.

If returned by the Service, this argument denotes the position of the next segment within the attribute value, as an offset in octets. If the segment just read was the last in the string, then this argument is set to zero. The result is present only if the Return Code result is OM_SUCCESS.

The value indicating the next position can be specified in a subsequent call as the position to start from, enabling sequential reading of the segments in a string value.

Elements

A space into which the Service returns the segment of the attribute value that you want to read. This argument is a string with two components, Elements and Length. Table 5–1 shows the initial values that you should give to these components.

Table 5–1 Initial Values for the Elements String

String Component	Initial Value
Elements	Pointer to a buffer
Length	The number of octets required to contain the segment that the function returns

You must make sure that the buffer is big enough to hold the number of octets.

The Service modifies the Elements argument. Each element that the function returns becomes an element in the string. The string's length becomes the number of octets actually required to hold the segment read (which may be smaller than the length initially specified.)

If the value of Local Strings is OM_TRUE, the final length of the string may not be the same as the initial length of the string. This depends on the characteristics of the translation into the local character set.

Chapter 7 gives details of the OM String data type.

Description

The function enables you to read a long string without requiring the Service to place a copy of the entire string in memory.

Return Values

OM_SUCCESS
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_OBJECT
OM_NO_SUCH_TYPE
OM_NOT_PRESENT
OM_NOT_PRIVATE
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR
OM_WRONG_VALUE_SYNTAX

om_remove

Removes and discards specified values of an attribute of a private object.

Syntax

OM_return_code om_remove (subject, type, initial_value, limiting_value)

Argument	Data Type	Access
subject	OM_private_object	read
type	OM_type	read
initial_value	OM_value_position	read
limiting_value	OM_value_position	read
return_code	OM_return_code	

C Binding

OM_return_code om_remove (subject, type, initial_value, limiting_value)

```
OM_private_object   subject,
OM_type             type,
OM_value_position   initial_value,
OM_value_position   limiting_value
```

Arguments

Subject

The private object from which you want to remove attribute values. The function does not affect the class of the subject.

Type

The type of the attribute from which you want to remove values. The type must not be OM_CLASS.

Initial Value

The position within the attribute of the first value to be removed.

If the value of Initial Value is OM_ALL_VALUES, or if it exceeds the number of values present in the attribute, the Service takes this argument to be equal to the number of values present in the attribute.

Limiting Value
The position within the attribute one beyond that of the last value to be removed. If this argument is less than the Initial Value argument, no values are removed.

If the value of Limiting Value is OM_ALL_VALUES, or if the value exceeds the number of values present in an attribute, the Service takes this argument to be equal to the number of values present in the attribute.

Description

This function removes and discards particular values of an attribute of a private object, the subject. If no values remain in an attribute after removal of the values you specify, the Service removes the attribute. If one of the values you specify is a subobject, the Service removes that value, and then applies `om_delete` to make the subobject inaccessible.

Return Values

OM_SUCCESS
OM_FUNCTION_DECLINED
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_OBJECT
OM_NO_SUCH_TYPE
OM_NOT_PRIVATE
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR

om_write

Writes a segment of a string to an attribute in a private object.

Syntax

OM_return_code om_write (subject, type, value_position, syntax, string_offset, elements)

Argument	Data Type	Access
subject	OM_private_object	read
type	OM_type	read
value_position	OM_value_position	read
syntax	OM_syntax	read
string_offset	OM_string_length	read-write
elements	OM_string	read
return_code	OM_return_code	

C Binding

```
OM_return_code  om_write  (subject, type, value_position, syntax, string_offset, elements )

OM_private_object     subject,
OM_type               type,
OM_value_position     value_position,
OM_syntax             syntax,
OM_string_length     *string_offset,
OM_string             elements
```

Arguments

Subject
The object into which you want to write the string segment.

Type
The type of the attribute to which you want to write the string segment.

Value Position
In a multi-valued attribute, the position of the value in which you want to place the string. This argument must have a positive value, and it must not exceed the number of values present in the attribute. If it equals the number

of values present, the Service inserts the segment at the end of the attribute as a new value.

Syntax

If you are writing a new value to an attribute, identify in this argument the syntax that you want the new value to have. It must be a permissible syntax for the attribute to which you are writing. To check that the syntax is permissible, consult the definition of the class of which the subject is an instance.

If you are overwriting or amending a value that is already present in the subject, the Service preserves the syntax of that value, so you can supply a null value.

String Offset

If supplied by the Client, this argument denotes the position, *p*, within the attribute value at which you want the first segment written. The position is specified as an offset in octets relative to the start of the string value.

If this argument has a value greater than the number of elements in the attribute value, the Service takes the argument to be equal to that number.

If returned by the Service, this argument denotes the position of the end of the last segment written. The position is specified as an offset in octets relative to the start of the string value.

The value returned by the Service as the position of the end of the last segment written can be specified as the position from which to start writing the next segment. This enables you to write segments sequentially.

Elements

The string segment that you want to write to the attribute, *n* elements in number. Copies of these elements occupy the positions, within the value, in the interval between *p* and *(p + n)*. The function discards any elements already in or beyond these positions.

The elements are bits, octets, or characters, depending on the nature of the string.

Description

This function writes a segment of an attribute value in a private object, the subject. The segment that the Service writes becomes the last segment in the attribute value. The function discards any segments in the attribute value whose offsets are equal to or greater than the offset specified in String Offset.

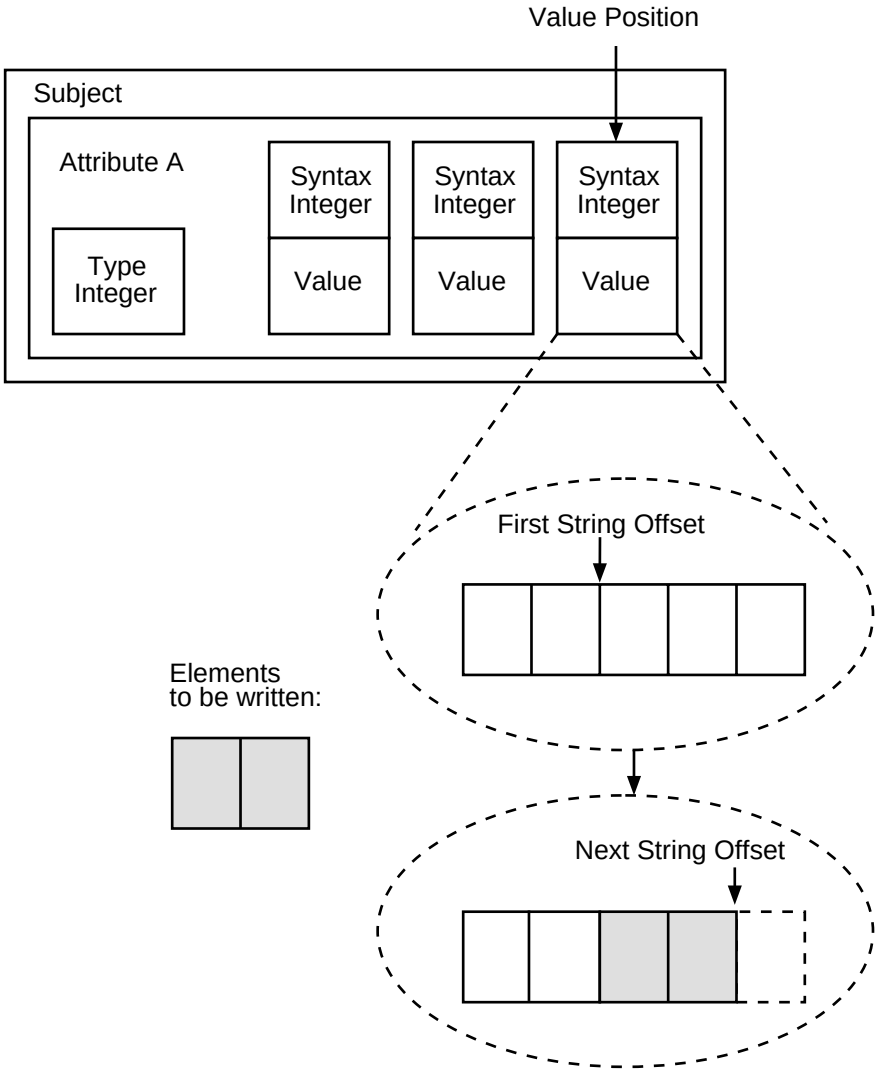
If the segment that the Service writes is in the local representation, the Service converts it to the non-local representation. This can result in loss of information and may result in a different number of elements than that specified.

Figure 5–5 gives an example of using `om_write`. In this example, Local String is assumed to be OM_FALSE. The diagram shows Value Position, First String Offset, and Next String Offset. It also shows that the element after Next String Offset is discarded.

Return Values

OM_SUCCESS
OM_FUNCTION_DECLINED
OM_FUNCTION_INTERRUPTED
OM_MEMORY_INSUFFICIENT
OM_NETWORK_ERROR
OM_NO_SUCH_OBJECT
OM_NO_SUCH_SYNTAX
OM_NO_SUCH_TYPE
OM_NOT_PRESENT
OM_NOT_PRIVATE
OM_PERMANENT_ERROR
OM_POINTER_INVALID
OM_SYSTEM_ERROR
OM_TEMPORARY_ERROR
OM_WRONG_VALUE_LENGTH
OM_WRONG_VALUE_MAKEUP
OM_WRONG_VALUE_POSITION
OM_WRONG_VALUE_SYNTAX

Figure 5–5 Example of Using the Write Function



6

OSI-Abstract-Data Manipulation Syntaxes

This chapter describes the syntaxes that OM attribute values can have.

The names of some OM syntaxes are constructed from syntax templates, as described in Section 6.1.

Section 6.2 gives a list of syntaxes defined for OM.

Section 6.3 gives more detailed information about the String syntax.

Most of the OM syntaxes correspond closely with ASN.1 data types and type constructors. Section 6.4 shows which syntaxes correspond to which ASN.1 data types.

6.1 Syntax Templates

A syntax template is a construction consisting of a primary identifier followed by an asterisk enclosed in parentheses. For example, String(*) is a syntax template with String as its primary identifier.

A syntax template represents a group of related syntaxes. If the text is referring to any member of the group, without distinction, it uses the primary identifier on its own. If the text is referring to a particular member of the group, it uses the template, and replaces the asterisk with one of a set of secondary identifiers associated with the template. For example, the String primary identifier might use the secondary identifier Printable to denote the Printable String syntax, thus giving String(Printable).

6.2 Syntaxes Defined for OSI-Abstract-Data Manipulation

The following syntaxes are defined for OM:

- **Boolean**

  A value of this syntax is a Boolean value; it may be either OM_TRUE or OM_FALSE.

- **Enumeration (*)**

  A value of any syntax that this template represents is one of a set of values associated with the syntax.

- **Integer**

  A value of this syntax is either a positive or a negative integer.

- **Null**

  There is only one value in this syntax; it is a placeholder used to denote that the attribute has no value.

- **Object (*)**

  A value of any syntax that this template represents is an object . The group represents all instances of any class associated with the syntax. The secondary identifier that distinguishes the member of this group of syntaxes is the name of the class.

- **String (*)**

  A value of any syntax that this template represents is a string of which the form and meaning are associated with the syntax.

  The secondary identifier that distinguishes the member of this group of syntaxes is usually the first word of the name of the ASN.1 type.

6.3 Strings

A string consists of a length indicator and an ordered sequence of zero or more elements. The elements may be bits, octets, or groups of octets representing characters. Strings containing these three types of elements are called bit strings, octet strings and character strings, respectively.

The length of a string is the number of octets in the string. A single character may be represented by more than one octet, therefore the length of a character string may not be equal to the number of characters it contains. Any constraint on the value length of a string is specified in the appropriate class definition.

The elements of a string are numbered. The position of the first element is zero. The positions of successive elements are successive positive integers. When passing a large string value across the interface, it may be necessary to segment it. A segment is any number of contiguous elements in a string. Segment boundaries have no semantic significance.

Figure 6–1 Structure of a String

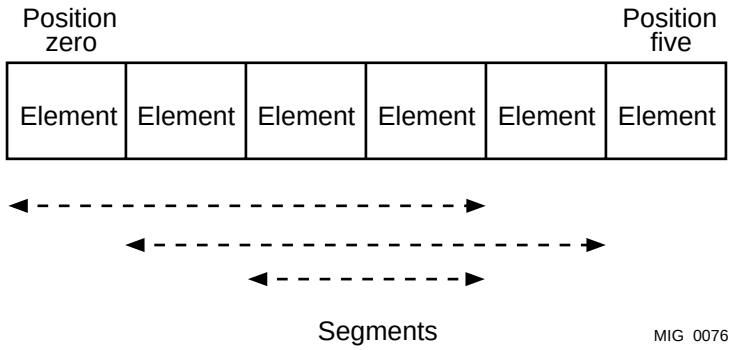


Figure 6–1 shows a string, with labelled elements, positions and segments.

Table 6–1 shows the secondary identifiers assigned to the syntaxes that form the String group.

Table 6–1 Secondary Identifiers of String Syntaxes

Bit String Identifiers	Octet String Identifiers	Character String Identifiers
Bit	Encoding[1]	General[2]
	Object Identifier[3]	Generalised Time[2]
	Octet	Graphic[2]
		IA5[2]
		Numeric[2]
		Object descriptor[2]
		Printable[2]
		Teletex[2]
		UTC Time[2]
		Videotex[2]
		Visible[2]

[1]The octets are those that the BER permit for the contents octets of the encoding of a value of any ASN.1 type.

[2]The characters are those permitted by the ASN.1 type of the same name. Values of these syntaxes must be in their BER encoded form.

[3]The values are those that the BER permit for the contents octets of the encoding of a value of the ASN.1 Object Identifier type.

The symbolic constants for the OM syntaxes are given in Appendix A.

6.4 OM Syntaxes and ASN.1

This section contains tables showing the relationships between OM syntaxes and ASN.1 data types and type constructors.

Table 6–2 Relationship of OM Syntaxes to ASN.1 Simple Types

ASN.1 Type	Functionally Equivalent OM Syntax
Bit String	String (Bit)
Boolean	Boolean
Integer	Integer
Null	Null
Object Identifier	String (Object Identifier)
Octet String	String (Octet)
Real	none

Table 6–3 Relationship of OM Syntaxes to ASN.1 Useful Types

ASN.1 Type	Functionally Equivalent OM Syntax
External	Object (External)
Generalised Time	String (Generalised Time)
Object Descriptor	String (Object Descriptor)
Universal Time	String (UTC Time)

Table 6–4 Relationship of OM Syntaxes to ASN.1 Character String Types

ASN.1 Type	Functionally Equivalent OM Syntax
General String	String (General)
Graphic String	String (Graphic)
IA5 String	String (IA5)
-	String (Local)
Numeric String	String (Numeric)
Printable String	String (Printable)
Teletex String	String (Teletex)
Videotex String	String (Videotex)
Visible String	String (Visible)

Table 6–5 Relationship of OM Syntaxes to ASN.1 Type Constructors

ASN.1 Type	Functionally Equivalent OM Syntax
Any	String (Encoding)
Choice	Object
Enumerated	Enumeration
Selection	*none* [1]
Sequence	Object
Sequence Of	Object
Set	Object
Set Of	Object
Tagged	*none* [2]

[1]This type constructor is used at specification time, and therefore has no corresponding syntax.

[2]The function of this type constructor is to distinguish the alternatives of a choice or the elements of a sequence or set. Use attribute types to perform this function.

There are ASN.1 type constructors other than those listed in Table 6–5. You can model the principal ASN.1 type constructors by using objects to group attributes, or by using attributes to group values. You can model these type constructors as classes of the following kinds:

- **Choice**

  Define an attribute type for each alternative, exactly one alternative being permitted in an instance of the class.

- **Sequence or Set**

  Define an attribute type for each sequence element or set element. If an element is optional, the attribute has either zero values or one value.

- **Sequence Of or Set Of**

  Define a single attribute with multiple values.

7

Object Management Data Types

This chapter describes the data types defined for OM. These are the data types that describe the format of OM attribute values, as determined by the syntax.

Table 7–1 lists all the OM data types.

Table 7–1 OM Data Types

Data Type	Description
Boolean	Boolean data value
Descriptor	Describes an attribute type and value
Enumeration	Enumerated data value
Exclusions	Exclusions parameter for the Get function
Integer	Integer data value
Modification	Modifications parameter for the Put function
Object	Handle to private or public object
Object Identifier	Object identifier data value
Private Object	Handle to an object in an implementation-defined representation
Public Object	Defined representation of an object that can be directly interrogated by a program
Return Code	A value returned from all OM functions, indicating that the function succeeded, or that it failed for a specified reason
String	Data value of String syntax
Syntax	Identifies a syntax type
Type	Identifies an OM attribute type

(continued on next page)

Table 7–1 (Cont.) OM Data Types

Data Type	Description
Type List	Enumerates a sequence of OM types
Value	Represents any data value
Value Number	Denotes the number of values for an attribute
Value Position	Denotes the position of a value within an attribute
Workspace	Identifies an application-specific API that uses OM, for example Directory or Message Handling

The definitions of the intermediate data types in the OM API are as follows:

```
typedef int             OM_sint;
typedef short           OM_sint16;
typedef long int        OM_sint32;
typedef unsigned        OM_uint;
typedef unsigned short  OM_uint16;
typedef long unsigned   OM_uint32;
```

OM_boolean

Type definition for a Boolean data value.

C Declaration

```
typedef OM_uint32 OM_boolean;
```

Description

A value of this data type is a Boolean value; it can be either *true* (represented by the constant OM_TRUE) or *false* (represented by the constant OM_FALSE).

OM_descriptor

Type definition for describing an attribute type and value.

C Declaration

```
typedef struct OM_descriptor_struct
{
    OM_type        type;
    OM_syntax      syntax;
    OM_value       value;
} OM_descriptor;
```

Description

A value of this data type embodies an attribute value. A sequence of descriptors (an array in C) can represent all the values of all the attributes of an object. A public object is composed of such a sequence of descriptors.

A descriptor has the following components:

Component	Type of the Component	Purpose
Type	OM_type	Identifies type of attribute value
Syntax	OM_syntax	Identifies syntax of attribute value
Value	OM_value	Specifies value of attribute

OM_enumeration

Type definition for an enumerated data value.

C Declaration

```
typedef OM_sint32 OM_enumeration;
```

Description

A value of this data type is an attribute value whose syntax is Enumeration.

OM_exclusions

Type definition for the Exclusions argument of the Get function.

C Declaration

```
typedef OM_uint OM_exclusions;
```

Description

A value of this data type is an unordered set of one or more values, all of which are distinct. Each value denotes an exclusion, as defined by the Get function. The following is a list of possible values:

- OM_EXCLUDE_ALL-BUT_THESE_TYPES
- OM_EXCLUDE_MULTIPLES
- OM_EXCLUDE_ALL_BUT_THESE_VALUES
- OM_EXCLUDE_VALUES
- OM_EXCLUDE_SUBOBJECTS
- OM_EXCLUDE_DESCRIPTORS
- OM_NO_EXCLUSIONS

Each value except OM_NO_EXCLUSIONS is represented by a distinct bit. The presence of a value is represented by one, and the absence of a value is represented by zero.

OM_integer

Type definition for an Integer data value.

C Declaration

```
typedef OM_sint32 OM_integer;
```

Description

A value of this data type is an attribute value whose syntax is Integer. Note that it is a signed integer.

OM_modification

Type definition for the Modifications argument of the Put function.

C Declaration

```
typedef OM_uint OM_modification;
```

Description

A value of this data type is an unordered set of one or more values, all of which are distinct. Each value denotes a modification, as defined by the Put function. The following is a list of possible values:

- OM_INSERT_AT_BEGINNING
- OM_INSERT_AT_CERTAIN_POINT
- OM_INSERT_AT_END
- OM_REPLACE_ALL
- OM_REPLACE_CERTAIN_VALUES

OM_object

Type definition for a handle to either a private object or a public object.

C Declaration

```
typedef struct OM_descriptor_struct *OM_object;
```

Description

A value of this data type represents a private or a public object. The value is an ordered sequence of one or more instances of the Descriptor data type. Refer to the descriptions of the OM_private_object and OM_public_object data types for details of constraints on the sequence.

OM_object_identifier

Type definition for an object identifier data value.

C Declaration

```
typedef OM_string OM_object_identifier;
```

Description

A value of this data type is an octet string that consists of the contents octets of the BER encoding of an ASN.1 object identifier.

The OM API provides some variables and macros for use with object identifiers:

- OM_C_<class_name>

  For each class name and object identifier, there is a global variable with the same name. For example, for the External class, there is a global variable called OM_C_EXTERNAL; for the object identifier for BER rules, there is a global variable called OM_BER. You can use these variables as arguments to routines, once they have been declared by the OM_EXPORT macro (see below).

- OM_OID_DESC(<type>, <OID_name>)

  This macro initializes a descriptor as part of a declaration (static initialization). The following table shows how the macro sets the components of the descriptor.

Component of Descriptor	Initial Setting
type	Type given in the call to the macro
syntax	OM_S_OBJECT_IDENTIFIER_STRING
value	OID_Name given in the call to the macro.

- OM_NULL_DESCRIPTOR

  This macro initializes a descriptor (as part of a declaration) to mark the end of a public object allocated by a Client.

- OM_EXPORT and OM_IMPORT

  The OM_EXPORT macro enables you to declare an object identifier. The OM_IMPORT macro allows you to make that object identifier available within a different compilation unit.

You must declare each object identifier used in your application program. You can declare an object identifier in the compilation unit that uses the object identifier, or you can declare all your object identifiers in the same compilation unit. The application program must explicitly reference an object identifier in every compilation unit that uses the object identifier.

The OM_EXPORT macro allocates memory for the constants that represent an object identifier. The following is an example of the use of the macro.

```
/* This code must appear in one compilation unit */

#include <xom.h>

OM_EXPORT(OM_C_ENCODING)
OM_EXPORT(OM_BER)
```

The following points apply to the OM_EXPORT macro:

- You may reference an OID (object identifier) or class name in the same compilation unit in which you use the OM_EXPORT macro to declare it.
- You must use OM_EXPORT to declare the storage for an OID, even if you only have a single compilation unit.

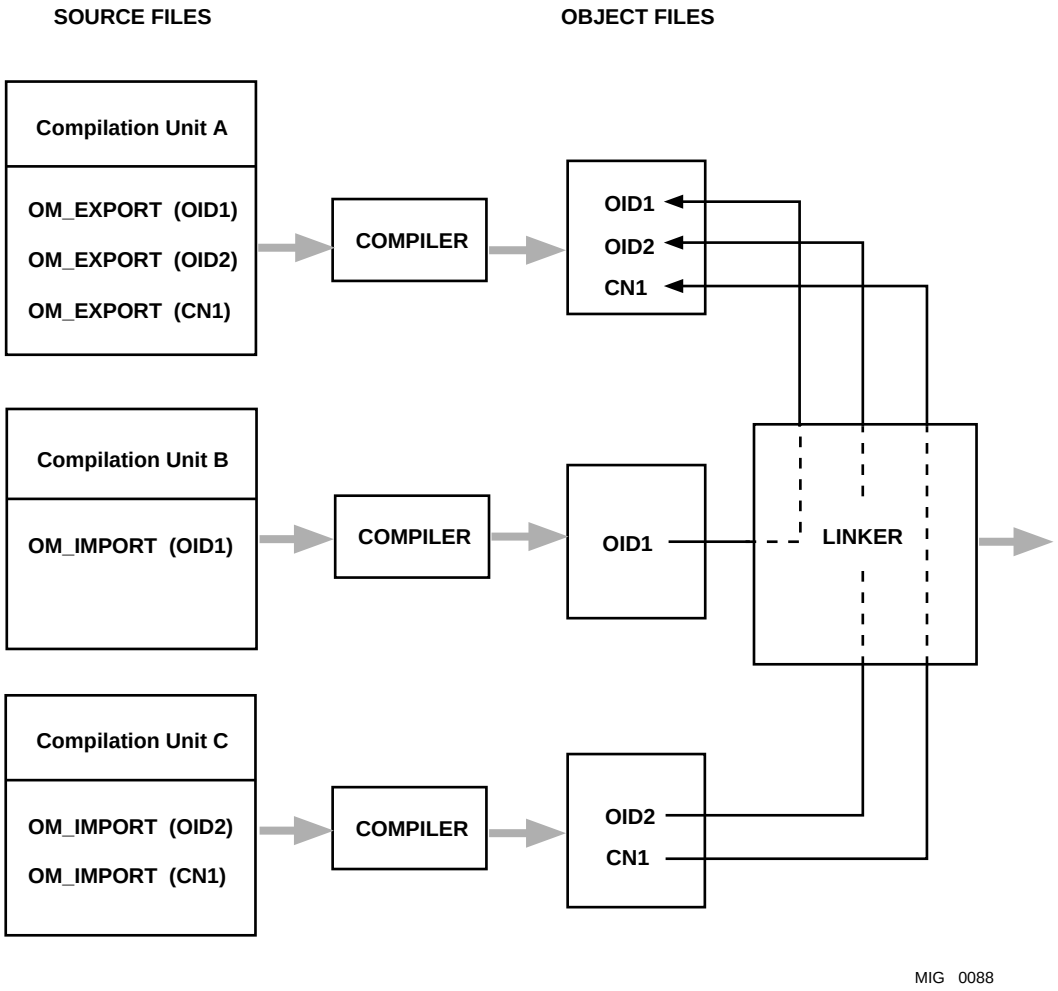
The following is an example of the use of the OM_IMPORT macro:

```
# include <xom.h>

main()

OM_IMPORT(OM_C_ENCODING)
OM_IMPORT(OM_C_BER)
{
/ Example 1  Define a public object of class Encoding
             Note: xxxx is a Message Handling Class
                   that can be encoded
 /

OM_descriptor my_public_object[] = {
    OM_OID_DESC(OM_CLASS, OM_C_ENCODING),
    OM_OID_DESC(OM_OBJECT_CLASS, MA_C_xxxx),
    { OM_OBJECT_ENCODING, OM_S_ENCODING, some_BER_value },
    OM_OID_DESC(OM_RULES, OM_C_BER),
    OM_NULL_DESCRIPTOR
    };

/ Example 2  Pass class Encoding as an argument to om_instance
 /

OM_return_code = om_instance(my_object, OM_C_ENCODING, &boolean_result)
}
```

OM_object_identifier

Figure 7–1 shows two object identifiers and a class name being exported from one compilation unit into two other compilation units.

Figure 7–1 Exporting and Importing Object Identifiers



OM_private_object

Type definition for a handle to an object in a private, implementation-defined representation.

C Declaration

```
typedef OM_object OM_private_object;
```

Description

A value of this data type is the handle or designator for a private object. Such a value consists of a single descriptor.

The components of this descriptor have the following values:

Component	Value
Type	private_object
Representation	Not used by Client
Syntax	Not used by Client
Value	Not used by Client

OM_public_object

Type definition for a defined representation of an object that can be directly interrogated by a programmer.

C Declaration

```
typedef OM_object OM_public_object;
```

Description

A value of this data type is a public object. Such a value consists of a sequence of one or more descriptors. All but the last of these descriptors represent attribute values of the object.

The last descriptor, OM_NULL_DESCRIPTOR, signals the end of the sequence of descriptors. The components of this descriptor have the following values:

Component	Value	Constant
Type	no-more-types	OM_NO_MORE_TYPES
Syntax	no-more-syntaxes	OM_NO_MORE_SYNTAXES
Value	Unspecified:	
	String.Length	OM_LENGTH_UNSPECIFIED
	String.Elements	OM_ELEMENTS_UNSPECIFIED

If an attribute has more than one value, the descriptors for those values must be adjacent to one another, in the same order as the values they represent.

The order in which attributes appear in the sequence is unspecified, with one exception. The Class attribute, if it is present, comes before any other attributes. The Syntax component of the descriptor for the Class attribute has the OM_SERVICE_GENERATED bit set if the object is created by the Service, or cleared if it is created by the Client.

The Digital OM API provides additional macros for creating public objects dynamically. These macros are in the header file xom.h and allow you to set an empty descriptor to your required value, once you have declared a public descriptor or descriptor list. The macros are as follows. Note that because they are proprietary extensions to the standard OM API, they have the prefix OMX.

- OMX_CLASS_DESC

  This macro sets an object's class descriptor to the specified class.

```
#define OMX_CLASS_DESC(desc,oidstr) { desc.type = OM_CLASS; \
                              desc.syntax = OM_S_OBJECT_IDENTIFIER_STRING; \
                              desc.value.string = oidstr; }
```

- OMX_BOOLEAN_DESC

  This macro sets a Boolean value descriptor for the specified type.

```
#define OMX_BOOLEAN_DESC(desc,type_name,bool) { desc.type = type_name; \
                                    desc.syntax = OM_S_BOOLEAN; \
                                    desc.value.boolean = bool; }
```

- OMX_ENUM_DESC

  This macro sets an enumerated value descriptor for the specified type.

```
#define OMX_ENUM_DESC(desc,type_name,enum) { desc.type = type_name; \
                                    desc.syntax = OM_S_ENUMERATION; \
                                    desc.value.enumeration = enum; }
```

- OMX_INTEGER_DESC

  This macro sets an integer value descriptor for the specified type.

```
#define OMX_INTEGER_DESC(desc,type_name,integr) { desc.type = type_name; \
                                    desc.syntax = OM_S_INTEGER; \
                                    desc.value.integer = integr; }
```

- OMX_OBJECT_DESC

  This macro sets an object descriptor for the specified type.

```
#define OMX_OBJECT_DESC(desc,type_name,obj) { desc.type = type_name; \
                                    desc.syntax = OM_S_OBJECT; \
                                    desc.value.object.object = obj; }
```

- OMX_OM_NULL_DESC

  This macro sets the null descriptor that terminates the descriptor list.

```
#define OMX_OM_NULL_DESC(desc) { desc.type = OM_NO_MORE_TYPES; \
               desc.syntax = OM_S_NO_MORE_SYNTAXES; \
               desc.value.string.length = 0;        \
               desc.value.string.elements = OM_ELEMENTS_UNSPECIFIED; }
```

- OMX_ATTR_TYPE_DESC

  This macro sets an attribute type descriptor using the specified object identifier string.

```
#define OMX_ATTR_TYPE_DESC(desc, ds_type, oidstr) { desc.type = ds_type; \
                           desc.syntax = OM_S_OBJECT_IDENTIFIER_STRING; \
                           desc.value.string = oidstr; }
```

- OMX_ZSTRING_DESC

  This macro sets a string descriptor given a null (zero) terminated string using the specified type name.

```
#define OMX_ZSTRING_DESC(desc,syntx,type_name,str) { desc.type = type_name; \
                         desc.syntax = syntx; \
      desc.value.string.elements = (void *)str; \
                         desc.value.string.length = strlen(str); }
```

- OMX_STRING_DESC

  This macro sets a string descriptor given the length and elements pointer and using the specified type name.

```
#define OMX_STRING_DESC(desc,syntx,type_name,str,len) {\
                             desc.type = type_name;\
                             desc.syntax = syntx; \
                             desc.value.string.elements = (void *)str; \
                             desc.value.string.length = len; }
```

The following is an example of the use of these macros.

```
/* Header files: */

   #include <xom.h>
   #include <xds.h>
   #include <xdsbdcp.h>

/* Declarations: */

   OM_EXPORT(DS_A_SURNAME)
   OM_EXPORT(DS_A_TITLE)

   OM_descriptor cpub_eis[5];

/* Assignment: */

   OMX_CLASS_DESC(       cpub_eis[0], DS_C_ENTRY_INFO_SELECTION);
   OMX_ATTR_TYPE_DESC(   cpub_eis[1], DS_ATTRIBUTES_SELECTED,DS_A_SURNAME);
   OMX_ATTR_TYPE_DESC(   cpub_eis[2], DS_ATTRIBUTES_SELECTED,DS_A_TITLE);
   OMX_ENUM_DESC(        cpub_eis[3], DS_INFO_TYPE, DS_TYPES_ONLY);
   OMX_OM_NULL_DESC(     cpub_eis[4]);
```

OM_return_code

Type definition for a value that all OM routines return. The value indicates either that the routine succeeded or that it failed for a specified reason.

C Declaration

```
typedef OM_uint OM_return_code;
```

Description

A value of this data type is an integer in the range [0, 2^{16}-1), which shows the outcome of an OM function.

In Chapter 5, each function description includes a list of the codes that may be returned by that function.

OM_string

Type definition for a data value of String syntax.

C Declaration

```
typedef OM_uint32 OM_string_length;
typedef struct
{
    OM_string_length   length;
    void               *elements;
} OM_string;

#define OM_STRING(string)
{
(OM_string_length)(sizeof(string)-1), (string)
}
```

Description

A value of this data type is an instance of a String syntax.

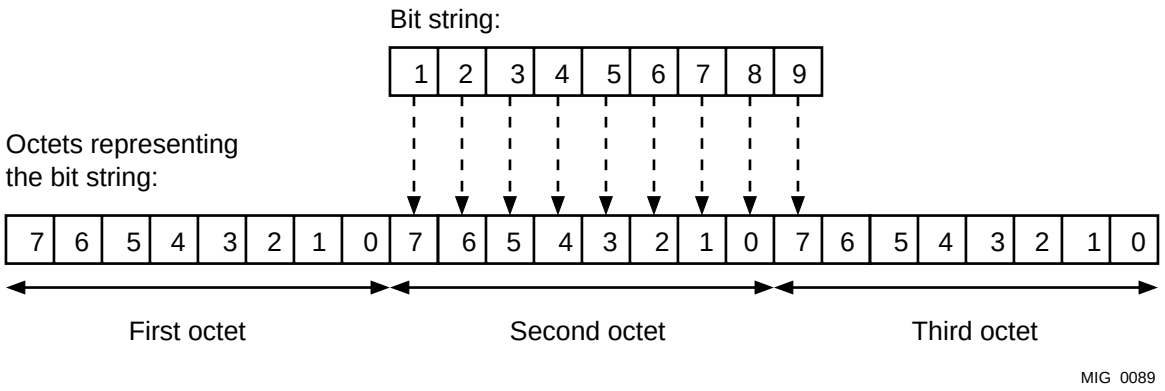
A string is represented as either a length-specified string or a null-terminated string. A string has the following components:

- Length
- Elements

  Bit Strings

  The bits of a bit string are represented as a sequence of octets. The first octet of the sequence stores the number of unused bits in the last octet of the sequence. The bits making up the string, starting with the most significant bit and ending with the least significant bit, occupy the second and consecutive octets, including as many bits of the final octet as are needed. Figure 7–2 illustrates this.

Figure 7–2 Representation of a Bit String in the C Interface



The diagram shows a bit string containing 9 bits. The first octet stores the number of unused bits in the last (third) octet, which in this case is 7.

Character Strings

The characters of a character string can be any sequence of octets acceptable as the primitive contents octets of the BER encoding of an ASN.1 type. The ASN.1 type defines the variety of character string. A zero character follows the characters of the character string. The length component of the string does not include this zero character. If the string is a null-terminated string, the zero character marks the end of the string.

The Service supplies a string value in the length-specified form. The Client may supply a string value to the Service in either the length-specified form or the null-terminated form.

The standard OM API provides a macro, {OM_STRING}, for constructing a value of the string data type when you know only the value of the elements component. The macro is for use with octet strings and character strings, but not with bit strings.

OM_syntax

Type definition for identifying a syntax type.

C Declaration

```
typedef OM_uint16 OM_syntax;
```

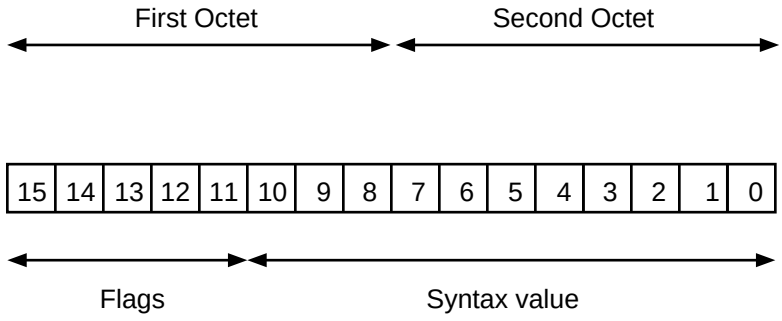
Description

A value of this data type is an integer in the range [0, 2^{16}) which denotes an individual syntax or a set of syntaxes.

The identifiers of all the OM syntaxes are given in Chapter 6.

The syntax component has six parts. Five of these parts are flags, and the sixth is the syntax value. The five flags are encoded in the five high-order bits of the syntax component. The syntax value is an 11-bit integer. This is shown in Figure 7–3.

Figure 7–3 Syntax Component of a Descriptor



To access the syntax value, use the OM_S_SYNTAX constant as shown in the following example:

```
OM_descriptor d;
if ((d->syntax & OM_S_SYNTAX) == OM_S_INTEGER)
    {process the value as an integer}
```

To access a flag value, use the constant for the flag, as shown in the following use of the OM_S_SERVICE_GENERATED constant:

```
OM_descriptor d;

if  (d->syntax & OM_S_SERVICE_GENERATED)
    {process the service-generated descriptor}
```

The flags are as follows:

Part	Bit Occupied	Constant
Long-String	15 (0x8000)	OM_S_LONG_STRING
No-Value	14 (0x4000)	OM_S_NO_VALUE
Local-String	13 (0x2000)	OM_S_LOCAL_STRING
Service-Generated	12 (0x1000)	OM_S_SERVICE_GENERATED
Private	11 (0x0800)	OM_S_PRIVATE

The Long-String bit is set if the component is a descriptor generated by the Service and the length of the descriptor value is greater than 1024.

The No-Value bit is set if the component is a descriptor generated by the Service, and the descriptor value is not present because a call to om_get specified OM_EXCLUDE_VALUES or OM_EXCLUDE_MULTIPLES.

Only one of the five flags can be set by the application, namely the Local-String flag. To set this flag, use the OM_S_LOCAL_STRING constant as shown in the following example:

```
OM_descriptor d;

d->syntax = d->syntax | OM_S_LOCAL_STRING
```

The Local-String bit is set if the component is a string represented in an implementation-defined local character set.

The Service-Generated bit is set if the component is a descriptor generated by the Service and is the first descriptor of a public object.

The Private bit is set if the component in the Service-generated public object or in the defined part of a private object contains a reference to the handle of a private subobject. (Note that this applies only when the descriptor is Service-generated; the Client need not set this bit in a Client-generated object containing a reference to a private object.)

OM_type

Type definition for identifying an OM attribute type.

C Declaration

```
typedef OM_uint16 OM_type;
```

Description

A value of this type is an integer in the range [0, 2^{16}-1), which denotes a type in the context of a package. The values *no-more-types*, *private-object*, and *public-object* have the meanings given to them by the OM_type_list, OM_private_object, and OM_public_object data types, respectively.

OM_type_list

Type definition for enumerating a sequence of OM attribute types.

C Declaration

typedef OM_type *OM_type_list;

Description

A value of this data type is an ordered sequence of zero or more type numbers, each an instance of the data type OM_type.

Another data value, no-more-types, follows and thus delimits the sequence. The C representation of the sequence is an array.

An example of the OM_type_list data type is given below.

```
OM_type rdn_type_list[2];

rdn_type_list[0] = DS_RDNS;
rdn_type_list[1] = OM_NO_MORE_TYPES;
```

OM_value

Type definition for representing any data value.

C Declaration

```
typedef struct
{
    OM_uint32     padding
    OM_object     object;
} OM_padded_object;
```

Format

```
typedef union OM_value_union
{
    OM_string               string;
    OM_boolean              boolean;
    OM_enumeration          enumeration;
    OM_integer              integer;
    OM_padded_object        object;
} OM_value;
```

Description

A value of this data type is an attribute value, and appears only as a component of a descriptor. You can determine what the syntax of the value is by examining the syntax component of the descriptor. Refer to the section on OM_descriptor for information on the structure of a descriptor.

A value of this data type has no components if either the syntax of the descriptor is OM_S_NO_MORE_SYNTAXES, or the No-Value bit in the syntax component is set.

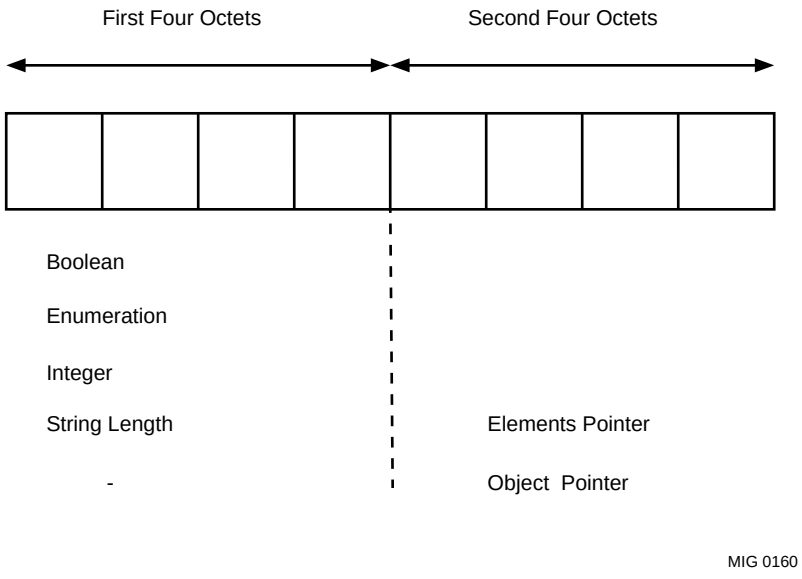
Otherwise, the value has exactly one component, determined by the value's syntax:

- A String if the syntax is a String syntax
- A Boolean if the syntax is Boolean
- An Enumeration if the syntax is Enumerated
- An Integer if the syntax is Integer
- An Object if the syntax is Object

When the variant in OM_value_union is a string, the Padding field in OM_ padded_object aligns the Elements component within the string with the Object field in the structure OM_padded_object. Both components contain pointers for the benefit of initialisation.

The way in which the OM_value data type is represented in memory is shown in Figure 7–4.

Figure 7–4 Representation of OM_value



OM_value_length

The number of bits, octets, or characters in a string.

C Declaration

```
typedef OM_uint32 OM_value_length;
```

Description

A value of this data type is an integer in the range [0, 2^{32}-1). It represents the number of bits in a bit string, or the number of octets in an octet string, or the number of characters in a character string. This data type allows you to define attribute constraints.

Note that the data type is not used when handling strings through the interface (see the section on the OM_string syntax).

OM_value_number

Type definition for denoting any constraints on the number of values for an attribute.

C Declaration

```
typedef OM_uint32 OM_value_number;
```

Description

A value of this data type is an integer in the range [0, 2^{32}-1). It denotes the number of values allowed for an attribute. This data type allows you to define attribute constraints.

OM_value_position

Type definition for denoting an attribute value's position within an attribute.

C Declaration

```
typedef OM_uint32 OM_value_position;
```

Description

A value of this data type is an integer in the range [0, 2^{32}-1), denoting the position of a value within an attribute. However, the value OM_ALL_VALUES has the meaning given to it by the Get function.

OM_workspace

Type definition for identifying an API that implements OM. For example, the workspace could be a Directory Service or a Message Handling Service.

C Declaration

```
typedef void *OM_workspace;
```

Description

A value of this data type is the Handle or Designator for a workspace, as returned by `ma_open`, `mt_open` and `ds_initialize`.

8

OSI-Abstract-Data Manipulation Header Files

The OM API requires two header files:

- xom.h

  This file defines the symbols that are available to client applications.

- xomi.h

  This file defines symbols for the C workspace interface.

One of the files, xom.h, includes the other by reference; you only need to include xom.h in your application.

After including xom.h, include the header files that are appropriate to the sort of application you are writing. For example, if you are writing an X.500 Directory Service application, you need to include the header files defined for the X.500 API, as descriibed in the X.500 API documentation. Whatever other header files you include, always include xom.h first.

To include xom.h use the following code:

```
#include <xom.h>
```

A
Symbolic Constants

This appendix lists the declarations that define the symbolic constants for the C interface of the OM API. The declarations are contained in the header file <xom.h>.

```
/* SYMBOLIC CONSTANTS */

/* Boolean */
#define OM_FALSE                                    ( (OM_boolean) 0 )
#define OM_TRUE                                     ( (OM_boolean) 1 )

/* Element Position */
#define OM_LENGTH_UNSPECIFIED                ( (OM_string_length) 0xFFFFFFFF)

/* Exclusions */
#define OM_NO_EXCLUSIONS                            ( (OM_exclusions) 0 )
#define OM_EXCLUDE_ALL_BUT_THESE_TYPES              ( (OM_exclusions) 1 )
#define OM_EXCLUDE_ALL_BUT_THESE_VALUES             ( (OM_exclusions) 2 )
#define OM_EXCLUDE_MULTIPLES                        ( (OM_exclusions) 4 )
#define OM_EXCLUDE_SUBOBJECTS                       ( (OM_exclusions) 8 )
#define OM_EXCLUDE_VALUES                           ( (OM_exclusions) 16 )
#define OM_EXCLUDE_DESCRIPTORS                      ( (OM_exclusions) 32 )

/* Modification */
#define OM_INSERT_AT_BEGINNING                      ( (OM_modification) 1 )
#define OM_INSERT_AT_CERTAIN_POINT                  ( (OM_modification) 2 )
#define OM_INSERT_AT_END                            ( (OM_modification) 3 )
#define OM_REPLACE_ALL                              ( (OM_modification) 4 )
```

```
#define OM_REPLACE_CERTAIN_VALUES          ( (OM_modification) 5 )

/* Return Codes */
#define OM_SUCCESS                         ( (OM_return_code) 0 )
#define OM_ENCODING_INVALID                ( (OM_return_code) 1 )
#define OM_FUNCTION_DECLINED               ( (OM_return_code) 2 )
#define OM_FUNCTION_INTERRUPTED            ( (OM_return_code) 3 )
#define OM_MEMORY_INSUFFICIENT             ( (OM_return_code) 4 )
#define OM_NETWORK_ERROR                   ( (OM_return_code) 5 )
#define OM_NO_SUCH_CLASS                   ( (OM_return_code) 6 )
#define OM_NO_SUCH_EXCLUSION               ( (OM_return_code) 7 )
#define OM_NO_SUCH_MODIFICATION            ( (OM_return_code) 8 )
#define OM_NO_SUCH_OBJECT                  ( (OM_return_code) 9 )
#define OM_NO_SUCH_RULES                   ( (OM_return_code) 10 )
#define OM_NO_SUCH_SYNTAX                  ( (OM_return_code) 11 )
#define OM_NO_SUCH_TYPE                    ( (OM_return_code) 12 )
#define OM_NO_SUCH_WORKSPACE               ( (OM_return_code) 13 )
#define OM_NOT_AN_ENCODING                 ( (OM_return_code) 14 )
#define OM_NOT_CONCRETE                    ( (OM_return_code) 15 )
#define OM_NOT_PRESENT                     ( (OM_return_code) 16 )
#define OM_NOT_PRIVATE                     ( (OM_return_code) 17 )
#define OM_NOT_THE_SERVICES                ( (OM_return_code) 18 )
#define OM_PERMANENT_ERROR                 ( (OM_return_code) 19 )
#define OM_POINTER_INVALID                 ( (OM_return_code) 20 )
#define OM_SYSTEM_ERROR                    ( (OM_return_code) 21 )
#define OM_TEMPORARY_ERROR                 ( (OM_return_code) 22 )
#define OM_TOO_MANY_VALUES                 ( (OM_return_code) 23 )
#define OM_VALUES_NOT_ADJACENT             ( (OM_return_code) 24 )
#define OM_WRONG_VALUE_LENGTH              ( (OM_return_code) 25 )
#define OM_WRONG_VALUE_MAKEUP              ( (OM_return_code) 26 )
#define OM_WRONG_VALUE_NUMBER              ( (OM_return_code) 27 )
#define OM_WRONG_VALUE_POSITION            ( (OM_return_code) 28 )
#define OM_WRONG_VALUE_SYNTAX              ( (OM_return_code) 29 )
```

```
#define OM_WRONG_VALUE_TYPE                    ( (OM_return_code) 30 )

/* String (Elements component) */
#define OM_ELEMENTS_UNSPECIFIED                ( (void*) 0 )

/* Syntax */
#define OM_S_NO_MORE_SYNTAXES                  ( (OM_syntax) 0 )
#define OM_S_BIT_STRING                        ( (OM_syntax) 3 )
#define OM_S_BOOLEAN                           ( (OM_syntax) 1 )
#define OM_S_ENCODING_STRING                   ( (OM_syntax) 8 )
#define OM_S_ENUMERATION                       ( (OM_syntax) 10 )
#define OM_S_GENERAL_STRING                    ( (OM_syntax) 27 )
#define OM_S_GENERALISED_TIME_STRING           ( (OM_syntax) 24 )
#define OM_S_GRAPHIC_STRING                    ( (OM_syntax) 25 )
#define OM_S_IA5_STRING                        ( (OM_syntax) 22 )
#define OM_S_INTEGER                           ( (OM_syntax) 2 )
#define OM_S_NULL                              ( (OM_syntax) 5 )
#define OM_S_NUMERIC_STRING                    ( (OM_syntax) 18 )
#define OM_S_OBJECT                            ( (OM_syntax) 127 )
#define OM_S_OBJECT_DESCRIPTOR_STRING          ( (OM_syntax) 7 )
#define OM_S_OBJECT_IDENTIFIER_STRING          ( (OM_syntax) 6 )
#define OM_S_OCTET_STRING                      ( (OM_syntax) 4 )
#define OM_S_PRINTABLE_STRING                  ( (OM_syntax) 19 )
#define OM_S_TELETEX_STRING                    ( (OM_syntax) 20 )
#define OM_S_UTC_TIME_STRING                   ( (OM_syntax) 23 )
#define OM_S_VIDEOTEX_STRING                   ( (OM_syntax) 21 )
#define OM_S_VISIBLE_STRING                    ( (OM_syntax) 26 )

#define OM_S_LONG_STRING                       ( (OM_syntax) 0x8000 )
#define OM_S_NO_VALUE                          ( (OM_syntax) 0x4000 )
#define OM_S_LOCAL_STRING                      ( (OM_syntax) 0x2000 )
#define OM_S_SERVICE_GENERATED                 ( (OM_syntax) 0x1000 )
#define OM_S_PRIVATE                           ( (OM_syntax) 0x0800 )
```

```
#define OM_S_SYNTAX                          ( (OM_syntax) 0x03FF )

/* Type */
#define OM_NO_MORE_TYPES                     ( (OM_type) 0 )
#define OM_ARBITRARY_ENCODING                ( (OM_type) 1 )
#define OM_ASN1_ENCODING                     ( (OM_type) 2 )
#define OM_CLASS                             ( (OM_type) 3 )
#define OM_DATA_VALUE_DESCRIPTOR             ( (OM_type) 4 )
#define OM_DIRECT_REFERENCE                  ( (OM_type) 5 )
#define OM_INDIRECT_REFERENCE                ( (OM_type) 6 )
#define OM_OBJECT_CLASS                      ( (OM_type) 7 )
#define OM_OBJECT_ENCODING                   ( (OM_type) 8 )
#define OM_OCTET_ALIGNED_ENCODING            ( (OM_type) 9 )
#define OM_PRIVATE_OBJECT                    ( (OM_type) 10 )
#define OM_RULES                             ( (OM_type) 11 )

/* Value Position */
#define OM_ALL_VALUES                ( (OM_value_position) 0xFFFFFFFF)
```

B

String Contents

This appendix lists the characters that are allowed in string types Printable, Numeric and IA5. A space is indicated by the word space.

B.1 Numeric Strings

The following characters are valid in numeric strings:

space
0 1 2 3 4 5 6 7 8 9

B.2 Printable Strings

The following characters are valid in printable strings:

space
' ( ) + , - . /
0 1 2 3 4 5 6 7 8 9
: = ?
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z

B.3 IA5 Strings

The following characters are valid in IA5 strings:

space
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9
: ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
[\ ] ^ _ ‘
a b c d e f g h i j k l m n o p q r s t u v w x y z
{ | } ~

C
Return Values

This appendix explains the return values used by the functions in the OM API. To find out which return values are used by a particular function, refer to Chapter 5.

Table C–1 OM API Return Values

Error	Meaning	OM_return _code
OM_FUNCTION_INTERRUPTED	The function was aborted by external intervention.	3
OM_MEMORY_INSUFFICIENT	There is not enough memory for the function to complete its task.	4
OM_NETWORK_ERROR	The Service cannot use the underlying network, and the link to the Client is lost or cannot be established.	5
OM_NO_SUCH_CLASS	The Client has passed an undefined class identifier to the Service.	6
OM_NO_SUCH_OBJECT	The Client has specified a nonexistent object, or an invalid handle for an object.	9
OM_NO_SUCH_SYNTAX	The Client has passed a public object to the Service with an undefined syntax identifier.	11
OM_NO_SUCH_TYPE	The Client has passed a public object to the Service with a type identifier that is not defined for the requested package.	12
OM_NOT_PRESENT	This error status indicates that an application built with the MAILbus 400 API has specified an incorrect recipient number in the Local NDR object passed to the Finish Delivery function.	16

(continued on next page)

Table C–1 (Cont.) OM API Return Values

Error	Meaning	OM_return_code
OM_NOT_PRIVATE	The Client has specified a public object when it should have specified a private one.	17
OM_PERMANENT_ERROR	The Service encountered an unspecified permanent problem.	19
OM_POINTER_INVALID	The Client has supplied an invalid pointer as a function argument.	20
OM_SYSTEM_ERROR	The Service cannot use the operating system.	21
OM_TEMPORARY_ERROR	The Service encountered an unspecified temporary problem.	22
OM_TOO_MANY_VALUES	An implementation limit prevents the addition to an object of another attribute value.	23
OM_WRONG_VALUE_LENGTH	The Client has passed an object to the Service with an attribute containing a value that violates the value length constraints for the attribute.	25
OM_WRONG_VALUE_MAKEUP	The Client has passed an object to the Service with an attribute containing a value that violates a constraint of the attribute's syntax.	26
OM_WRONG_VALUE_NUMBER	The Client has passed an object to the Service with an attribute containing a value that violates the value number constraints for the attribute.	27
OM_WRONG_VALUE_SYNTAX	The Client has passed an object to the Service with an attribute containing a value with an illegal syntax.	29
OM_WRONG_VALUE_TYPE	The Client has passed an object to the Service containing a value of the wrong type for the class.	30

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W

X